An Operationalized Model for Defining Computational Thinking

Tony Lowe
Purdue University Department of Engineering Education
West Lafayette, IN, USA
lowe46@purdue.edu

Sean Brophy
Purdue University Department of Engineering Education
West Lafayette, IN, USA
sbrophy@purdue.edu

***Abstract*— The Computational Thinking (CT) conceptual framework is entering its second decade of research yet still lacks a cohesive definition by which the field can coalesce. The lack of clear definition makes assessment tool challenging to formulate, pedagogical efforts difficult to compare, and research difficult to synthesize. This paper looks to operationalize differing definitions of CT enhancing the ability to teach then assess the presence of CT. Expanding upon CT definitions, industry practices and processes, and educational theory, we link existing concepts and propose a new element to model an active definition of CT as a theoretical framework to guide future research. Our model updates existing CT definition by formally including Modeling, introducing Socio-Technical processes, separating Information Gathering from Data Collection and adding emphasis to Testing as a vital CT concept. We feel these elements and interconnections make CT is easier to describe and measure.**

Keywords—Abstraction, Computational Thinking, Modeling, Testing, Debugging

I. Introduction

This research paper synthesizes current literature on Computational Thinking (CT) to provide an operationalized model of CT concepts for use in assessment and pedagogy. To promote CT we must have a clear definition of the key concepts and competences to include, and defined in such a way that we can assess its presence in learners. When Wing initially presented the concept **[1]** she proposed a generalized definition which later she refined [2] and others expanded upon [3]–[5]. While literature agrees on several key ideas, gaps still exist in the scope of CT as well as dissatisfaction and how it is defined [6]. Wing proposes that CT is not just about learning to program, but is an activity conducted within many professions. One study finds the number of people involved in managing computational solutions is double the number of CS professionals [7], not considering the countless people who define and work within processes supported by automation. The full potential of CT is unleashed when non-programmers understand the potential of computing to accelerate daily tasks and the pace of discovery. Educators need clear definitions and ideally developmental trajectories in order to convey the complex ideas within CT to novice programmers and non-programmers alike. The inexactness of the current definitions makes measuring mastery much less designing effective instruction difficult.

A. Issues in Assessing Computational Thinking

The lack of standardized assessments of CT is a hurdle to understanding its presence and growth in learners [4], [8]. While many proposals for assessing CT exist [5], [9], [10], they either focus on a very limited aspect of CT (e.g. sequencing) or are tied to activities for those on a pre-programming track. Computational thinking is more than programming, thus an assessment which measures growth in programming aspects of CT concepts alone has not historically transferred to other problem solving contexts [11]. If we only focus on learners on a programming track, we are unlikely to meet teach "CT for all" [1]. CT skills outside programming contexts demands we teach and assess CT concepts that transfers to domains outside programming.

When assessment is rooted in programming it is difficult to 'see' CT in other fields. It is easier to come up with programming tasks demonstrating CT, but examples do exist that use computational devices other than computers. The Jones Live-Map system was constructed in 1909 as a navigational device, using the car's odometer to provide real-time driving directions [12]. The Live-Map design includes an abstraction of the roads to be navigated on a route, collects input from gears, translates the input using rules and represents data using an arrow pointing at the next driving instruction to be undertaken. The entirely mechanical system seems to meet the exact criteria for applying computational thinking, yet realized without any microprocessor or code. Do we have assessment tools that could measure the presence of CT within such a design?

B. Challenges in Teaching Computational Thinking

Computational Thinking is a complex domain with a large number of highly integrated concepts, as we will demonstrate later. Each learner comes to CT with different prior experience in each CT concept. Some have experience with programming, while others are novices in code. Some have experience finding patterns, testing, or design, while others may only have a heard of each. Jerome Bruner models learning as the development of enactive representations (e.g. a mental model of how the world works), followed by ikonic representations (e.g. using rough pictures to communicate the concepts), and finally symbolic representations (e.g. in this case, formal use of modeling languages or source code) [13], [14]. Bruner's model is salient to CT as computational devices recreate 'the laws of Physics' in the architecture of processors and operating systems. Computer 'physics' are

978-1-5090-5920-1/17/$31.00 ©2017 IEEE

more flexible and yet more precise than the physical world. From infancy, we create an enactive understanding how, for instance, gravity behaves. Schools later teach us how to generalize "stuff falls downwards" and model gravity using physics diagrams and math equations. The symbolic formulas (mostly) back our understanding of how gravity behaves in the real world. Computational systems do not typically display 'visible attributes' such as gravity to develop enactive representations, in fact designers go to great lengths to make interactions with computers mimic the real world. A computer can easily redefine defy the laws of gravity in simulation, but has a very hard time understanding basic language. Computers can distort our enactive understanding of the physical world in simulation, but still are built upon an immutable set of rules which we must learn. Humans can communicate very imprecisely to each other and still achieve acceptable results. A software solution must be realized perfectly to obtain even simplest results from a computer. Most computational solutions require logical modeling of the rules to be realized where the brain often relies on intuition in decision making [15]. Thinking computationally may in fact be 'unnatural' unless specifically taught as many of our decision making processes lie outside traditional logical reasoning.

Computational Thinking education often starts by presenting ikonic or even symbol representations without first teaching 'how computation works'. We present the rules of computers and describe high level concepts, sometimes before showing the problems to be solved. The definition of CT also suffers from a 'top down' approach where, in our experience, even highly trained professionals struggle to differentiate the finer details of some esoteric concepts included within CT. Replacing concepts with a working model of CT may better serve learners and educators.

C. Research Goal

Our goal in this paper is to operationalize the CT concepts and competencies in a way that allows novices easier access to complex concepts and enables more direct assessment of learning. CT concepts tend to be defined as static lists of skills with standalone definitions. Instead we hope to show how concepts collaborate and interact as a related process. We will build a concept map from the elements of CT using literature, data, and a dash of instinct from decades of experience building computational systems and teaching Computer Science (CS). Our model documents how the consensus CT concepts naturally integrate, fills gaps where they do not, and draws out tacit elements which binds the concepts in a more visible manner.

II. Defining Computational Thinking

A. Concepts and Competencies from Literature

Literature defines and redefines CT, offering multiple lists of concepts and competencies which support and expand on each other. We chose three primary sources to give a breadth of potential topics included in CT. The first, which forms the foundation for the model presented in this paper, is a synthesis of work by the International Society for Technology in Education (ISTE), Computer Science Teacher Association (CSTA), and hosted by Google [3]. Their definition includes eleven key ideas which are supported by the other two sources, Grover and Pea[4] who collect their definition from a literature review and Brennan and Resnick[5] who look at CT from more of a pre-coding perspective. We identified 25 distinct concepts proposed from these three sources which can be condensed into 9 primary categories as follows.

- Abstraction
- Decomposition
- Patterns (Recognition and Generalization)
- Algorithms
- Data (Collection, Analysis, and Representation)
- Parallelism
- Iteration
- Simulation (and Automation)
- Testing and Debugging

Experts familiar with CT most likely resonate with this list and see the emerging CT definition, yet some of these concepts are elusive, particularly to novices learners or educators. In this section we will describe the conventional and competing definitions in support of the model/framework to be presented later.

1) Abstraction, Patterns, and Decomposition

Abstraction might be the most important concept in CT based on is prevalence in literature, but it also may be the most inconstantly defined. Within CS, abstraction has taken on two distinct uses. The first is commonly used in CT as "identifying and extracting relevant information to define main idea(s)" [3, p. 1]. This use of abstraction is to hide away details which are not required to understand a greater concept as often relates to encapsulation [16]. The other common use of abstraction however relates to inheritance and uses the fewer details as an extension point for future functionality [17]. The first view of abstraction simply hides details not relevant to a particular viewpoint, where the second requires a larger understanding of the context and future needs. They are both clearly 'abstraction' but require different levels nuance in instruction of expertise in action.

The multiple definitions of abstraction can also be seen in other disciplines. Philosophers debate abstraction centering on how abstractions are identified *and used*. Hans Radder defined extensibility, the ability to extend the abstraction to a new domain, as the distinguishing characteristic of an abstraction [18]. Extensibility aligns well with the Object-Oriented representation of abstraction in inheritance. Nancy Cartwright counters Radder by defining abstraction as a set of rules, behaviors, or characteristics, or as she states "A's do X". This basic rule covers the idea of simplification as abstraction, but asks for more than simply leaving out information. It demands the abstraction to be defined in some way as having rules, not simply excluding details, and Cartwright does not stop there. Abstraction become clear through contextualizing rules in a context demonstrating specific behavior, or "In *I*, A's do X". A good abstraction should be more than leaving

out details how 'a thing' is implemented, but should define rules that transcend the current implementation to the next. We can see this in Object-Orientation as objects are concretized implementations of a specific Class, which could be considered a contextualized abstraction. The OO class is an abstraction which describes a concept to be modeled, where the object a single 'avatar' of that concept which does work. Philosophers' views of abstraction provide a definition deeper yet present in modern programming paradigms, but not clearly present in CT.

Identifying and using patterns is vital in defining abstractions. Cohen says "data abstraction is that of defining a pattern for objects… [which] can inherit all of the attributes defined by the pattern" [16, p. 31]. An abstraction is more than hiding details of a full implementation, instead capturing key details which will later be represented in computational processes. Simple abstractions can be formulated through observation: What does each customer have in common when they make an order? What steps are always taken in the manufacturing of the product? This process is the CT definition of Pattern Recognition, where the application of trends is Pattern Generalization. Patterns and abstractions are intertwined as abstractions are formed by recognizing then generalizing patterns.

Abstraction and decomposition are opposite approaches to analysis in some sense. Abstractions capture patterns from the real world reducing details to capture what is common. Decomposition "is breaking down data, processes, or problems into smaller, manageable parts" [3, p. 2]. Decomposition looks at a big complex thing and breaks it up into simpler parts by capturing or imagining details of how it is or can be accomplished. Abstraction starts with the problem in action and works bottom-up to reduce detail and find commonality, while decomposition starts with a high level problem and works top-down to capture details. Both are vital, yet opposing approaches in software development history. Early software design highly utilized decomposition, where big tasks (e.g. Build a report) are broken down into manageable steps (e.g. Gather data, perform calculations, format printout). Object-Oriented Analysis initially focuses less on decomposition and more of capturing the actors and objects of the system, utilizing abstraction to capture the essential data and behaviors to group into Classes. Decomposition exists within the Object-Oriented approach, and abstraction helps form reusable, modular components in structured design making these concepts difficult to separate. Decomposition, abstraction and patterns clearly seem intertwined concepts for describing computational systems.

2) Algorithms and Data

Algorithms may be the most agreed to CT concept across literature: "Algorithms are tools for developing and expressing solutions to computational problems" [4, p. 39]. The term is so ubiquitous that it often goes undefined, but generally is thought to be a list of steps to complete a task. Often unsaid is the inextricable tie to data and context within an algorithm. In CS, algorithms are typically encased within a procedure, function, or method that allows for inputs and outputs but is tied into the context provided within the application architecture. Non-computational algorithms examples are often given as fixed set of commands without context, for example, a recipe. A recipe provides a list of the needed inputs (ingredients) and the functional steps to transform inputs into the desired output, but does not elaborate on the context for cooking. The recipe still assumes the context of a kitchen with the required tools. It demands shared understanding of predefined abstractions such as "simmer", "beat", or "fold". The concept of an algorithm is simple, but precisely describing a computational algorithm requires assumptions on known abstractions and the representations of data.

The representation and manipulation of data is core in computational systems. From a highly programming-centric view, Brennan and Resnick define data as "involv[ing] storing, retrieving, and updating values" [5, p. 6]. The framework from Google breaks "Data" into three components: to collect (or gather), analyze (find patterns), and represent (or visualize) [3]. These definitions are unclear however, in who exactly is performing each action? Computer systems can gather data, use algorithms to interpret meaning and take further action entirely on their own. Likewise the general public can gather data (from computers!) and make decisions and act. Computational thinkers also gather data about a problem space, look for patterns, and define data collection systems and algorithms. Part of the goal of CT is to recognize this symbiosis, yet learners need to understand the role of computer and people within any given system, and differentiate form the process of building computational systems.

3) The rest of the elements

The remaining concepts are important but in many ways subordinate to the ones already covered. Parallel processing is an important realization of systems, but mainly a deployment approach. Some solutions are better in parallel but few require it, making it optional in early learning of CT concepts. Simulation is an interesting concept, both by a computer but also when considered as part of design. Simulation is "developing a model to imitate real-world processes" [3, p. 2]. A simulation can be a role-play or other human-centered exercise to better understand tradeoffs and perform early testing when designing a system or software is an automated form of simulation. Automation only appears as a concept in the CSTA/ISTE/Google CT framework, but is implied in the other definitions of simulation. For non-programmers, knowing CT can be simplified as harnessing the power of simulation, with or without automation.

Iteration, or iterative and incremental approach, is a difficult concept to integrate with other CT concepts depending on its definition. It is not discussed in the Google hosted framework, but is in each of the other sources. Brennan and Resnick [5] talk about partially as an approach within algorithm design similar to parallelism. More often it is defined as approach to the design of a computational (or any) system, where early trials inform later design choices. In this context, all aspects of CT can include iterative

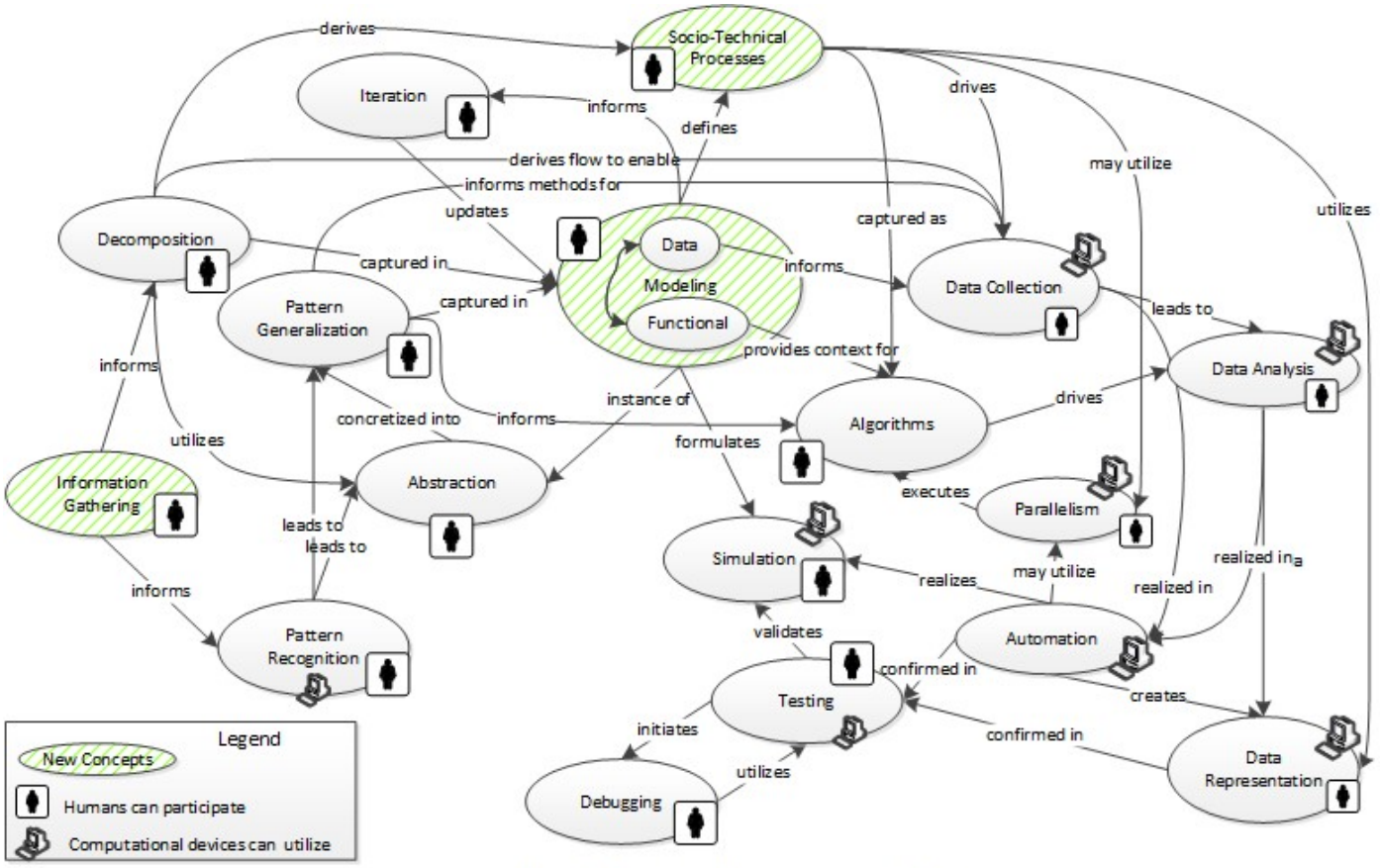

Figure 1 Operationalized concept map for CT

approaches, yet it is difficult to capture any instance of this, unless you are documenting the development process.

We feel Testing and Debugging are vital CT concept and skills excluded from the Google hosted framework. Testing is the process of removing errors from a system and ensuring it fulfills the user needs. When the system does not behave as expected, debugging follows to determine and remediate the failure. Neither of these concepts is new or exclusive within CT, but may be taken to new extremes in computers and has the potential drive learning and assessment.

B. Operationalizing the Computational Thinking Definition

The major shortcoming in understanding CT is a clear vision of how the defining elements interrelate. To bridge this gap we are creating a concept map (Figure 1) to define relationships between concepts and capture implicit relationships. We have extended our concept map to include context clues 'who' utilizes a concept. When the human silhouette appears, it implies that task is (or can be) completed by humans, where the computer icon indicates a task typically performed by a computational device. By modeling the CT elements in a concept map, we can see both the inherit complexity of CT but also begin to break down into a learning progression to introduce CT to learners.

The nine main concepts taken from the three frameworks described above are present in our operational model for CT with their core relationships. The next sections will walk through the concept map, but as a few notes may be helpful to setup a few additions to the nine main elements. We have expanded *Patterns* and *Data* using the sub-concepts presented in [3] to help demonstrate how the concepts are applied in conjunction. Automation from [3] is included to distinguish between work by human and computational devices in end products. Finally, we have added three concepts to capture implicit ideas which help bind stated concepts: *Modeling*, *Socio-Technical Processes,* and *Information Gathering* (distinguished from Data Collection).

1) Abstraction, Patterns, and Decomposition

Abstraction, Patterns, and Decomposition are perhaps the most 'problematic' aspects of defining CT, but literature does give hints to how they relate. Considered independently, it is easy to become muddled in the many ways each concept is used. In our operational model we hope to focus on a primary use and hint at alternatives. The most common reference to CT concepts is in the design of a system not its execution[1]. Focusing on the left side of Figure 1, people gather

[1] In is important to note, pattern recognition can certainly be the goal of an algorithm within automation. Computers can look for faces in pictures, grammatical errors, or any sort of pattern as well. We are not ignoring this aspect of pattern recognition or other CT concept in automation, but trying to distinguish the human design activities from the end user/automated system.

information (elaborated on later), from which *Pattern Recognition* may emerge. Those patterns can be generalized into common behaviors, or even to abstractions which must be concretized into patterns in order to be captured in design. An *Abstraction* by itself is interesting, but most useful when formulated into an approach to solving problems, as occurs in *Pattern Generalization*. CT literature implies that all of this information eventually lands in a *Model*. The model documents *Decomposition* and the patterns generated. Abstractions appear as contextualized instances within the model. The summation of the design portion of these three key concepts is captured in the model as will be discussed.

2) Modeling, Data, and Algorithms

The concept of modeling underlies many of the other CT concepts and regularly appears in the literature. In a paper devoted to abstraction, Kramer states "Modeling is the most important engineering technique"[19, p. 41] . Sooriamurthi states "Programming is a process of modeling a world" [20, p. 2]. Wing refers to modeling [1] even in the initial work, yet it seems to be taken for granted as a CT skill. Including modeling in the concept map binds many of the otherwise esoteric concepts. The *Model* becomes the place where *Abstraction*, *Decomposition* and *Patterns* are documented. It describes the connection between information and processing forging a bridge between *Data* and *Algorithms*. The model captures the context in which data is stored and provided to algorithms. Models become the artifact of design which can be assessed by experts or even simulated manually enabling early *Testing* and *Debugging*. Modeling helps capture complexity in a manageable form. Making modeling explicit as a core element of CT has the potential to accelerate both assessment and learning.

The model moves intangible concepts to tangible, capturing mental representation in physical form. Many assessments of CT are limited to concepts such as control structures [21] in order to manage the complexity. While many CT assessments opt for multiple-choice driven analysis questions, models provide a place where assessment can occur within design tasks. Instead of reciting a definition of abstraction, the learner's model can be explored for presence of abstraction, particularly outside the 'expected answer'. The model becomes a point of mentoring where the learner's mental representation is presented and a mentor can provide feedback on representational accuracy, application of CT concepts, or even process and thinking approaches. Rather than being told the 'right answer' to a multiple choice question, the learner can being to see the continuum of approaches within CT as well as their maturing conceptual understanding.

Simulated execution of models provides experiential opportunities to 'see' how design choices impact systems. The model captures the intended behavior of the system or process being designed. Models document the user's story/system's flow which can be evaluated even without needing to produce a real system. Lee et al. state that "analysis is a reflective practice that refers to the validation of whether the abstractions made were correct." [8, p. 33]. This suggests learning may be better when learners model their abstractions and evaluate validity through simulation, refining their model based on the findings. This human form of simulation, entirely without computers, can support instruction through finding and fixing problems in the model, the very process of iterative and incremental development. As their model matures, learners witness iteration in action by seeing how their documented model changes over time. Modeling 'cleans up' the links between CT concepts as well as providing an artifact to capture and assess a learner's growing knowledge of CT.

3) Socio-Technical Processes

Nearly every computational system is a part of a larger human-centered process. Even satellite systems charged with exploring remote areas of our solar system, designed to function with no direct human interaction, still receive guidance and report data back to people. Each concept within CT frameworks hint at this larger purpose, but adding how people interact with computational systems feels vital. Complex systems are decomposed into tasks, some handled by humans and others automated in computational devices. When we model systems, they are capturing the boundaries between computational components and people. Designers use *Information Gathering* to inform the model, but *Socio-Technical Processes* also need data. *Data Collection* can be completed by people or computers, analyzed and eventually reported back as a future step in the process (discussed further in the next section). Capturing the interaction between machine computation and human cognition provides context to the greater purpose of CT beyond making computer programs.

Socio-technical processes are likely much easier to model and comprehend. It is impossible to introduce novices to all the aspects of a computer program, but much more reasonable to walk through a human-centered process. It is easier to develop an enactive representation [13] CT in action by demonstrating an abstraction or module (e.g. using a spreadsheet to enter data and see a result) without requiring the management or manipulation code. Lee et al. recommend introducing students using a "use-modify-create" approach [8] where learners first act as users, then modify an existing program, before taking on full creation. For domains where programming is not the end goal, simulation of a socio-technical process could develop a similar enactive understanding of CT concepts and the value of non-programmers in CT activities. For all learners it is important to remember that computational systems stem from human processes, and benefit from the input of many types of people.

4) Data and Information Gathering

The ultimate purpose of any computational system is to gather, process and present data, yet the use of data is a small portion of and in many ways vaguely defined in CT. Grover and Pea [4] almost casually include data as a concept while Brennan and Resnick [5] define it more as variables in programs than a key aspect of computation. The Google hosted framework [3] takes the time to break the use of data into three key stages, but each definitions does not provide

clarity on the role of people versus devices. *Data Collection* is defined as "gathering information" [3, p. 1], which could relate both to the human process of gaining understanding of the system, or the task (human or computer) of gathering input from sources for a process, or both. This ambiguity likely leads new teachers and learners into misconceptions about CT. To that end, we have separated *Information Gathering* from *Data Collection* as a unique CT concept.

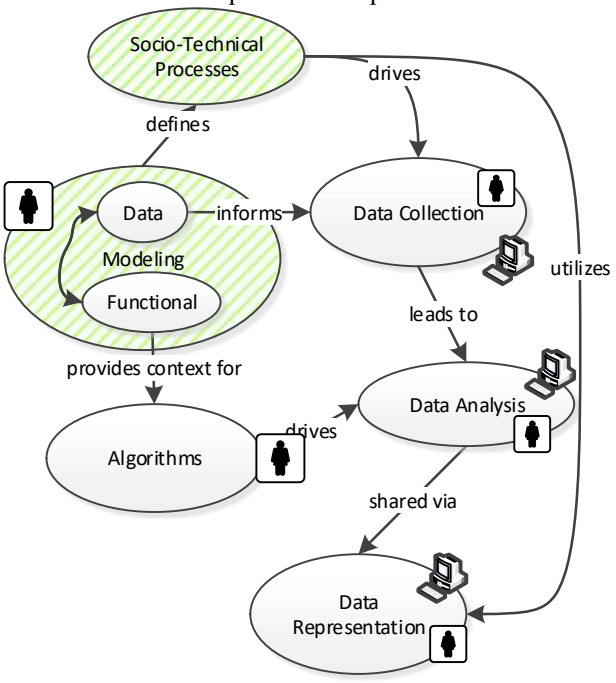

Figure 1 Breakout of CT Data concepts

Gathering information is a human process to better understand the role of a computational system. Beyond CT, information gathering is a key aspect of any design process. Any STEM or design education likely includes instructions how to seek context and stakeholder needs. Conjoining the traditional information gathering process with the specialized data input solutions of CT seems to add confusion rather than clarity. Designers gather information to design what data the computational system needs to collect from the real world? Bundling *Data Collection* and *Information Gathering* into one concept, creates a confusing circular dependency! By separating these concepts, Information Gathering feeds into the human processes of design and modeling, which then in turn defines the Data Collection as a piece of the end design. This seems more transparent than having Data Collection (the problem scoping facet) informing Data Collection (the inputs to the solution facet) and forcing learners and novice educators to tease out the differences.

The three CSTA/ISTE/Google *Data* phases show the transformation of data within a CT system. Collected data is only the starting point, as broken out from the full concept map in Figure 1, provided to *Data Analysis* processes informed by the *Algorithms* derived from the *Functional* and *Data Models*. The results of *Data Analysis* are shared in a final representation of data used to communicate knowledge back to people participating in the larger *Socio-Technical Process*. The Socio-Technical process is codified in the model, and drives *Data Collection* whether by device or human intervention, as full analysis and representation of data could also be conducted by people, processors or both.

In our operationalized model, *Data Analysis* is a separate idea from the design activities which would include *Abstraction*, *Patterns* and *Decomposition*. *Data Analysis* instead is performed by *Algorithms* defined within the *Model* that may have an algorithmic goal of finding patterns in real-world data, but is bound to the context of solving the problem defined within the *Socio-Technical Process*, rather than the open ended goal of defining a computational system.

Finally, *Data Representations* are vital parts of computational systems to report information in a way that is valuable to users. How information is presented can have a large impact on how processes function. Many learners may never develop the skills to create data representations, but can at least witness power of custom representations and their role in serving the larger process being served by automated solutions.

5) Empahsizing Testing

Testing is a vital part of delivering systems of any nature. In software, testing's importance can be seen through numbers: it often exceeds a third of the budget of a software project [22] and one organization alone has certified more than a half million testers [23]. People who participate in testing compose a large group of non-programmers who have a strong interest in understanding CT. *Testing* and *Debugging* appeared in two of the three CT frameworks, being left out of the framework developed primarily by educators [3]. This could be in response to not being "code centric" but seems a massive oversight in scoping CT.

Testing affords an entry point to CT concepts requiring little prior experience. Starting with testing follows the Lee et al.'s pattern of "use-modify-create" [8] , staring with 'using' a system. Testing requires learners to form an expectation how the system will respond given a set of inputs, making it an active evaluation of the system rather than passively following instructions. Learning starts by evaluating the boundaries of the 'problem' and anticipating how the computational system will behave. For instance, a very popular early computational thinking challenge is navigating a maze [21], [24]–[26]. This task emphasizes the sequencing of steps within algorithms, but also demands strong spatial reasoning skills to complete. Young people often struggle with "which way to turn" more than stringing together a set of commands. By focusing first on 'testing' a planned route, we can first teach the task of navigating the maze before taking on CT concept of forming an algorithm. A test case can 'break', showing the route to be 'broken' either due to a bad sequence (algorithm problem) or a wrongly reasoned turned (spatial reasoning issue). Learners can be tasked with testing activities to understand a problem space (how do I navigate a maze?), discovering how their computational device behaves at an enactive level (what happens when I press this turn button?), before taking the next

step of introducing debugging to analyze algorithm for problems (does this route work?). By testing first we can learn the rules of the game (the left and right buttons turn us 90º but do not move us forward!) separated from the tricky task of developing algorithms or other CT concepts.

The combination of testing and debugging can accelerate student learning. Jerome Bruner suggests learning improves by "shielding a learner from distraction, by forefronting crucial features of a problem, by sequencing the steps to understanding, by promoting negotiation, or by some other form of 'scaffolding' the task at hand" [27, p. 69]. Student learning is slowed when too much is going on and they must stop to detangle complexity. Testing provides novices with a focused engagement with computational systems with scaffolding to remove distractions. The learner does not need to modify the system itself yet, but simply modify the inputs to scientifically test behavioral change. The learner's mental representation of the problem space and computational solution can be gradually updated with evidence-based experience rather than abstract rules of 'how concepts should work'.

For instance, a young learner could start with a set of incorrect instructions to navigate a robot through the maze. The 'failure' is caused by a "right turn" when a left turn is needed; representing a common misconception in spatial reasoning. Since the student did not create the algorithm, they are less invested in proving their initial guess is correct, and can instead focus on what is actually occurring rather than what they expect should happen. Now when the student 'debugs' the instructions they start by using then continue to modifying the 'algorithm' either recognizing the error or uncovering and correcting their misconception (e.g. 'left' and 'right' are relative to the robot's new position). The learner's only responsibility is to uncover the problem step, rather than solving the entire problem. This scaffolding approach allows the learner to focus on first the problem space (spatially navigating a maze) before the computational element (designing and 'coding' an algorithm). Testing and debugging provide a highly authentic activity easily introduced early in the learning process to build enactive representations upon which to build.

III. Applying the CT Model

1) Assessment

Assessment of CT should start with the learner's ability to translate *Information Gathering* skills into *Modeling*. A model provides the most tangible representation of growing knowledge and exposes the presence of other CT concepts such as *Abstraction*, *Patterns*, and *Decomposition*. The model binds and contextualizes *Data* and *Algorithms* in both computational devices and also *Socio-Technical Processes*. In early stages students can evaluate their solution (*and skill*) by 'executing' their model within a manual simulation. The model documents a learning trajectory while authentically demonstrating iterative design through synergistic formative assessment. Learners can receive feedback on their model, including both the solution space and reflective feedback on the CT concepts. Having learners create models provides more authentic summative assessment than memorized definitions or forcing non-programming track learners to code. Using models provides a flexible approach to problem solving that can be translated into different domains. CT learning can be measured in projects unrelated to coding, by allowing the specification of computational devices but never requiring their actual design or implementation. Including *Information Gathering* as a starting point, *Modeling* as a point of capture, and defining *Socio-Technical Processes* to allow for non-programming related context, simplifies CT concepts to allow for more relevant assessment for all learners.

2) Pedagogy

Modeling provides a tangible artifact for assessment but does little on its own to promote learning. In fact, learning a modeling language may add a similar burden as learning a programming language initially. While a modeling language does not demand programming's perfect syntax, if too free-form models are difficult to interpret and becomes useless for broad assessment. We propose the key to modeling pedagogy is heavy scaffolding integrating CT concepts, yet allowing learners to focus on learning each concept in turn.

Testing provides a low-demand introduction to CT concepts. A successful test case allows learners to experience a system as intended and develop and enactive understanding of how the problem domain and computational device solve a particular need. By introducing controlled failures in predefined test cases, we focus student's attention on specific aspects of the problem space and CT concepts in turn, while expanding enactive representations. Testing is followed by reading premade models to build ikonic representations of the computational system being tested and teaching modeling syntax. By introducing change in stakeholder needs, the learner can see changes in test cases and model maturing the enactive and binding to ikonic representation the learner is developing. When CT concepts are formally introduced, the learner will already hold a 'gut feeling' how they behave in the sample system, linking the 'abstract CT concepts' to a 'concretized solution'. This approach maintains a low demand for programming skill level, making CT plausible for diverse subjects and learners. A test system could be a highly configurable software product, or even be a human driven process with instructions modified on paper. Testing allows the focus to shift from programming skills to CT concepts much earlier in the learning curve.

IV. Future Directions

This paper is the culmination of literature, initial research, and anecdotal experience, but is very much a starting point. A great deal of time and energy has gone into expanding the body of research on CT, but we do no feel a definition has emerged which facilitates consistent assessment or a learning trajectory for learners from early grades through college level programing courses. Which aspects of CT are pre-cursors to CT-for-all versus mostly coding-centric? What level of expertise do different learners need to achieve? This framework does not answer all of these questions, but

hopefully provides a framework for how we can define empirical research. Future questions could include:

- Can modeling demonstrate CT skills development?
- Can testing and debugging as an initial introduction to CT improve the development of CT/coding skills?
- Does developing CT concepts and competencies make learners more likely to participate in careers that involve CT?
- What CT skills are most often employed in non-programming domains and thus are most valuable to teach to pre-college learners?

We feel the model presented in this paper can help to inform future research by providing an operationalizing several definition of CT beyond a set of related but unlinked concepts. By continuing a conversation on what is CT, hopefully research and assessment techniques can better document development of CT in learners of all ages and backgrounds.

ACKNOWLEDGMENT

This work is sparked from many discussions with our peers at Purdue's INSPIRE while working on our STEM+C project (NSF Award Number 1543175) focusing on K-2 learners. Having a multi-disciplinary team researching computational thinking exemplifies the need and value of CT-for-all and the worthiness of sharing these ideas to kids and adults from all walks of life.

REFERENCES

[1] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[2] J. M. Wing, "Computational Thinking," in *Microsoft Research Asia Faculty Summit*, 2012.

[3] Google, "Computational Thinking Concepts Guide." [Online]. Available: https://docs.google.com/document/d/1i0wg-BMG3TdwsShAyH_0Z1xpFnpVcMvpYJceHGWex_c/edit.

[4] S. Grover and R. Pea, "Computational Thinking in K-12: A Review of the State of the Field," *Educ. Res.*, vol. 42, no. 1, pp. 38–43, 2013.

[5] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," *Annu. Am. Educ. Res. Assoc. Meet. Vancouver, BC, Canada*, pp. 1–25, 2012.

[6] E. Jones, "The Trouble with Computational Thinking," 2011. [Online]. Available: https://c.ymcdn.com/sites/www.csteachers.org/resource/resmgr/JonesCTOnePager.pdf.

[7] "The Hidden Half," *Change the Equation*, 2015. [Online]. Available: http://www.changetheequation.org/blog/hidden-half. [Accessed: 01-Jan-2017].

[8] I. Lee, F. Martin, J. Denner, B. Coulter, W. Allan, J. Erickson, J. Malyn-Smith, and L. Werner, "Computational thinking for youth in practice," *Acm Inroads*, vol. 2, no. 1, pp. 32–37, 2011.

[9] M. Israel, Q. M. Wherfel, S. Shehab, E. A. Ramos, A. Metzger, and G. C. Reese, "Assessing collaborative computing: development of the Collaborative-Computing Observation Instrument (C-COI)," *Comput. Sci. Educ.*, vol. 3408, no. December, pp. 1–26, 2016.

[10] D. Weintrop and U. Wilensky, "Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs," *Int. Comput. Educ. Res. Conf.*, no. January, pp. 101–110, 2015.

[11] R. D. Pea, "Logo Programming and Problem Solving," *Conf. Pap.*, vol. 150, no. ir 014 383, pp. 1–10, 1983.

[12] N. Paumgarten, "Getting There: The science of driving directions.," *The New Yorker*, Apr-2006.

[13] J. S. Bruner, "On cognitive growth," in *Studies in cognitive growth: A collaboration at the center for cognitive studies*, Wiley and Sons, 1966, pp. 1–29.

[14] J. S. Bruner, "On cognitive growth II," in *Studies in cognitive growth: A collaboration at the center for cognitive studies*, Wiley and Sons, 1966, pp. 30–67.

[15] D. Kahneman, *Thinking, fast and slow*. Macmillan, 2011.

[16] A. T. Cohen, "Data abstraction, data encapsulation and object-oriented programming," *SIGPLAN Not.*, vol. 19, no. 1, pp. 31–35, 1984.

[17] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Comput. Surv.*, vol. 17, no. 4, pp. 471–523, 1985.

[18] S. F. Martı, "Epistemic Groundings of Abstraction and Their Cognitive Dimension *," vol. 78, no. 3, pp. 490–511, 2016.

[19] J. Kramer, "Is abstraction the key to computing? Abstraction : What is it? Why is it so important?," *Commun. ACM*, vol. 50, no. 4, pp. 37–42, 2007.

[20] R. Sooriamurthi, "The essence of object orientation for CS0: concepts without code," *J. Comput. Sci. Coll.*, pp. 67–68, 2010.

[21] A. Mühling, A. Ruf, and P. Hubwieser, "Design and First Results of a Psychometric Test for Measuring Basic Programming Abilities," *WiPSCE '15*, 2015.

[22] C. Saran, "Application testing costs set to rise to 40% of IT budget," *Computer Weekly*, 2015.

[23] "Facts & Figures." [Online]. Available: http://www.istqb.org/about-as/facts-figures.html.

[24] M. Roman-Gonzalez, J. C. Perez-Gonzalez, and C. Jimenez-Fern??ndez, "Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test," *Comput. Human Behav.*, 2016.

[25] L. A. Gouws, K. Bradshaw, and P. Wentworth, "Computational thinking in educational activities," *Proc. 18th ACM Conf. Innov. Technol. Comput. Sci. Educ. - ITiCSE '13*, p. 10, 2013.

[26] E. Kazakoff and M. Bers, "Programming in a robotics context in the kindergarten classroom: The impact on sequencing skills," *J. Educ. Multimed. Hypermedia*, vol. 21, pp. 371–391, 2012.

[27] J. S. Bruner, "Celebrating divergence: Piaget and Vygotsky," *Hum. Dev.*, vol. 40, no. 2, pp. 63–73, 1997.