Educ Inf Technol (2013) 18:351–380
DOI 10.1007/s10639-012-9240-x

Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework

Pratim Sengupta · John S. Kinnebrew · Satabdi Basu · Gautam Biswas · Douglas Clark

Published online: 3 January 2013

Abstract Computational thinking (CT) draws on concepts and practices that are fundamental to computing and computer science. It includes epistemic and representational practices, such as problem representation, abstraction, decomposition, simulation, verification, and prediction. However, these practices are also central to the development of expertise in scientific and mathematical disciplines. Recently, arguments have been made in favour of integrating CT and programming into the K-12 STEM curricula. In this paper, we first present a theoretical investigation of key issues that need to be considered for integrating CT into K-12 science topics by identifying the synergies between CT and scientific expertise using a particular genre of computation: agent-based computation. We then present a critical review of the literature in educational computing, and propose a set of guidelines for designing

P. Sengupta
Mind, Matter & Media Lab, Vanderbilt University, Nashville, TN, USA

P. Sengupta (✉) · D. Clark
Department of Teaching & Learning, Peabody College, Vanderbilt University, Nashville, TN, USA
e-mail: pratim.sengupta@vanderbilt.edu

D. Clark
e-mail: doug.clark@vanderbilt.edu

J. S. Kinnebrew · S. Basu · G. Biswas
Department of EECS/ISIS, Vanderbilt University, Nashville, TN, USA

J. S. Kinnebrew
e-mail: john.kinnebrew@vanderbilt.edu

S. Basu
e-mail: satabdi.basu@vanderbilt.edu

G. Biswas
e-mail: gautam.biswas@vanderbilt.edu

D. Clark
Learning, Environment & Design Lab, Vanderbilt University, Nashville, TN, USA

learning environments on science topics that can jointly foster the development of computational thinking with scientific expertise. This is followed by the description of a learning environment that supports CT through modeling and simulation to help middle school students learn physics and biology. We demonstrate the effectiveness of our system by discussing the results of a small study conducted in a middle school science classroom. Finally, we discuss the implications of our work for future research on developing CT-based science learning environments.

Keywords Computational thinking · Agent-based modeling and simulation · Visual programming · Multi-agent systems · Learning by design · Computational modeling · Science education · Physics education · Biology education

1 Introduction

Wing (2006, 2008) and others (National Research Council 2010) have described computational thinking as a general analytic approach to problem solving, designing systems, and understanding human behaviors. While computational thinking (CT) draws upon concepts that are fundamental to computing and computer science, it also includes practices such as problem representation, abstraction, decomposition, simulation, verification, and prediction. These practices, in turn, are also central to modeling, reasoning and problem solving in a large number of scientific and mathematical disciplines (National Research Council 2008).

Although the phrase "Computational Thinking" was introduced by Wing in 2006, earlier research in the domain of educational technology also focused on similar themes, e.g., identifying and leveraging the synergies between computational modeling and programming on one hand, and developing scientific expertise in K-12 students on the other. For example, Perkins and Simmons (1988) showed that novice misconceptions in math, science and programming exhibit similar patterns in that conceptual difficulties in each of these domains have both domain-specific roots (e.g., challenging concepts) and domain general roots (e.g., difficulties pertaining to conducting inquiry, problem solving, and epistemological knowledge). Complementarily, Harel and Papert (1991) argued that programming is reflexive with other domains, i.e., learning programming in concert with concepts from another domain can be easier than learning each separately. Along similar lines, several other researchers have shown that programming and computational modeling can serve as effective vehicles for learning challenging science and math concepts (Guzdial 1995; Sherin 2001; Hambrusch et al. 2009; Blikstein and Wilensky 2009; diSessa 2000; Kaput 1994; Kynigos 2007).

However, despite these synergies, computational thinking and programming have not been integrated with K-12 science curricula in any significant way (National Research Council 2010). In this paper, we address this issue with a particular focus on integrating CT with scientific modeling in K-12 classrooms. With respect to science education, our paper is grounded in the *science as practice* perspective (Duschl 2008; Lehrer and Schauble 2006; National Research Council 2008). In this perspective, the development of scientific expertise is inseparably intertwined with the development of epistemic and representational practices (e.g., Giere 1988; Lehrer and Schauble 2006; Nersessian 1992; National Research Council 2008). In this perspective,

modeling is veiwed as the "language" of science (Giere 1988), and is therefore identified as the core scientific representational practice. *Our central hypothesis is that the development of scientific modeling in K-12 curricula can be synergistically supported by a science curriculum that is based on computational thinking.*

The motivation for our work is twofold. First, from the perspective of instructional design (i.e., development of instructional technologies and curricular practices), previous research shows the following: a) integrating computational modelling and programming with K-12 science and math curricula can be challenging due to a high teaching overhead and the challenges students face in learning programming (Sherin et al. 1993); and b) the design of programming-based learning environments needs to be rethought for integration with science education (Guzdial 1995; diSessa et al. 1991a, b; diSessa 2000; Sengupta 2011). Second, from the perspective of children's development of scientific expertise, it has been established by researchers that developing scientific reasoning and expertise requires sustained, immersive educational experiences. This is particularly reflected in the recent efforts to develop long-term, multi-year learning progressions for science in K-12 classrooms (Lehrer et al. 2008; Corcoran et al. 2009). Therefore, integrating CT with science and mathematics in a manner that supports the development of students' scientific expertise requires the design of coherent curricula in which computational thinking, programming, and modeling are not taught as separate topics, but are interwoven with learning in the science domains. Along similar lines, the ACM K-12 Taskforce (2003) also recommends integrating programming with curricular domains such as science and math, rather than teaching programming as a separate topic at the K-12 levels. However, there exists no theoretical framework that can support such an integration, or address the challenges peratining to instructional design as identified at the beginning of the previous paragraph. Our goal in this paper is to establish such a framework.

In establishing this framework, we first propose the following four components of the framework:

1. *Relationship between CT and Scientific Expertise:* In Sections 2.1 and 2.2, we explicitly identify the synergies between CT and scientific modeling;
2. *Selection of a Programming Paradigm:* In Sections 2.3 and 2.4, we provide justifications for the choice of a particular programming paradigm—agent-based computation—and a mode of programming—visual programming—in order to facilitate scientific modeling alongside CT;
3. *Selection of Curricular Science Topics:* In Section 2.5, we outline the rationale behind choosing an initial set of topics (kinematics and ecosystems) in science that are amenable to our technology, but at the same time illustrate the breadth and generality of our approach;
4. *Principles for System Design:* In Section 3, we elaborate the design principles for integrating CT and Science Learning, based on which we designed the CTSiM learning environment. These include: a) supporting low-threshold as well as high-ceiling learning activties; b) design of programming primitives, c) supporting algorithm visualization; and d) sequencing learning actitivities in a constructivist fashion.

Following our presentation of the framework, we show how such a theoretical framework can be applied to developing a computer-based learning environment in Section 4, where we present CTSiM (Computational Thinking in

Simulation and Modelling), a visual-programming based learning environment for middle school science. In this section, we present the system architecture, as well as the elements of the user interface, that were designed in order to implement the theoretical framework. In Section 5, we describe the curricular modules that we developed to support the integration of CT and science learning. Finally, in Section 6, we provide empirical evidence of the effectiveness of our proposed theoretical framework and the CTSiM learning envrionment, based on results from a pilot study conducted with 6th grade students.

2 Setting the stage: Computational thinking, educational computing and K12 science

2.1 Abstractions in computational thinking and scientific expertise

Given the centrality of the notion of *abstractions* in computational thinking (Wing 2008), it is imperative for us to understand the relationships between abstractions in CT and in scientific expertise. To do so, we first begin with a quick historical account of how the notion of *abstractions* has been studied by philosophers, which in turn influenced developments in psychology and education. As we shall see, Wing's (2008) notion of abstractions, as well as the connection between abstractions in scientific inquiry and CT bears upon some of the ways in which abstractions have been studied historically.

Early Greek philosophers such as Plato (360 BCE/2003) and Aristotle (384 BCE) were concerned with understanding the relationship between *forms* (i.e., abstract qualities such as *beauty, equality, sameness, difference*) and *sensibles* (i.e., the realm of perceptual sensations). Later philosophers such as Locke (1690/1979) viewed abstraction as a mental process, and accordingly framed the discussion of the abstract-concrete distinction within the mind. Locke proposed two types of ideas: *particular* and *general*. Particular ideas are constrained to specific contexts in space and time. General ideas are free from such restraints and thus can be applied to many different situations. In Locke's view, abstraction is the process in which "ideas taken from particular beings become general representatives of all of the same kind" (Locke 1690/1979). In the field of psychology, as Von Glaserfield pointed out, Jean Piaget also developed theories of cognitive development centered on the notion of abstractions that bear deep similarities to Locke's (see Von Glaserfeld 1991).

Wing (2006) defined the phrase "computational thinking" to indicate a "*thought process involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent*" (Wing 2006, p 1). According to her, the "nuts and bolts" in computational thinking involve dealing with abstractions in the following ways: a) defining abstractions, b) working with multiple layers of abstraction, and c) understanding the relationships among the different layers (Wing 2008). Abstractions, according to Wing (2006), give computer scientists the power to scale and deal with complexity. In order to provide examples of how abstractions are used in computer science, Wing (2006) noted:

> *Abstraction is used in defining patterns, generalizing from instances, and parameterization. It is used to let one object stand for many. It is used to*

capture essential properties common to a set of objects while hiding irrelevant distinctions among them. (Wing 2006, p1).

Wing's conceptualization of abstraction, as the excerpt above shows, therefore, emphasizes the notion of *generalization*: abstractions, in her view, are *generalized* computational representations that can be used (i.e., applied) in multiple situations or contexts. In this sense, her definition of abstraction is similar to Locke's.

However, Wing (2006) further argued that in their practice, computational scientists and engineers use abstractions *recursively*, which in turn leads them to work with multiple *layers* of abstraction. For example, Wing (2006) pointed out that an algorithm is an abstraction of a process that takes inputs, executes a sequence of steps, and produces outputs to satisfy a desired goal. However, designing efficient algorithms inherently involves designing abstract data types. An abstract data-type, in turn, defines a set of values and operations for manipulating those values, hiding the actual representation of the values from the user of the abstract data type. An example of such a recursive abstraction (i.e., multi-layered abstraction) is the application programming interface (API) of a software component, and the underlying layer of the component's implementation (Wing 2008, 2006). Similarly, Schmidt (2006) points out that software researchers and developers typically engage in creating abstractions that help them program in terms of their contextualized design goals (e.g., the specific problem that they are solving), rather than the underlying computing environment (e.g., CPU, memory, and network devices), and shield them from the complexities of these environments. Computation, thus typically requires working simultaneously with at least two, and usually more layers of abstraction (Wing 2008). Wing also argued that the abstraction process necessarily involves design thinking, i.e., it requires computational scientists and engineers to decide what details need to be highlighted and represented (and complementarily, what details can be ignored) in each layer, as well as the connection (i.e., the "fit") between different layers. This is also echoed by other researchers who investigate design thinking in software engineering (Ho 2001; Cross 2004, etc.).

The discussion above suggests that although Wing (2006) defined computational thinking as a "*thought process*", computational thinking becomes evident only in particular forms of epistemic and representational practice that involve the generation and use of external representations (i.e., representations that are external to the mind) by computational scientists. From a pedagogical perspective, this is an important point. As educators, our work is grounded in a general pedagogical framework, which suggests that students learn best when they engage in *design-based learning* activities that focus on the design and consequential use of external representations for modeling and reasoning (Papert 1980, 1991; Kolodner et al. 2003; Edelson 2001; Blikstein and Wilensky 2009; Sengupta et al. 2012). This is also aligned with the "*science as practice*" perspective (Lehrer and Schauble 2006; Duschl 2008; National Research Council 2008), as discussed in the Introduction section of this paper. Therefore, to support the development of students' computational thinking, we believe that we need to engage students in the process of developing the computational representational practices that Wing (2008, 2006) identified to be the "nuts and bolts" of computational thinking.

Given that our pedagogical goal is to integrate these representational practices with scientific inquiry in K-12 classrooms, it is imperative that we now identify the role

Table 1 Correspondences between abstractions in computational thinking and scientific inquiry

Computational thinking	Scientific inquiry
Encapsulation of functionality and state within an object/interface that provides a meaningful abstraction while hiding implementation details	Creating coherent, formal representations of scientific processes and phenomena; Understanding functions as dynamic objects that can be used to represent multiple phenomena
Classes or **Agent-breed** as a type or category of objects/agents (defining their possible behavior and properties) that can be instantiated as individuals, which act and change state independently	Agent-based thinking (e.g., micro-level reasoning in physics; reasoning about individual-level agents in a multi-agent biological system)
Class inheritance and **polymorphism** to reuse functionality and properties among a set/category of related classes, while allowing specializations in each and substitution of any specialized version as an instance of the more general category	Classification and hierarchical organization; Biological taxonomies and inheritance
Distributed problem-solving, self-organization, and swarm intelligence in **decentralized** and **multi-agent systems**	Macro-level reasoning; Generating aggregate-level equations and functional forms from individual-level variables; Emergent and swarm behavior in social scientific and biological systems
Algorithm design and **complexity analysis**, including formal representations and reasoning	Mechanistic reasoning and explanations; Generating formal models (e.g., mathematical equations; models; functions; etc.) to analyze scientific phenomena
Iterative and incremental development as a formal **software engineering** process	Iterative theory refinement through modeling; Model matching; Model refinement
Test-driven software development, such as **unit testing**, and software verification	Hypothesis testing; Iterative refinement of models; Verification and validation of models

that *abstractions* play in scientific inquiry. Similar to computational thinking, in the domain of scientific inquiry, we also pertain to Locke's definition of abstraction that emphasizes *generalization* of representations and ideas beyond specific situations, and *generalizability* as an attribute of scientific representations, such as theories and models. This can be understood as follows. At the broadest level, we argue that the process of scientific inquiry involves the generalizable practice of generating models, which themselves are generalizable mathematical and formal representations of scientific phenomena. Note that scientific inquiry can take many forms—observational, comparative, or theoretical; and it can be conducted in many contexts—physics laboratories, astronomical observatories, or biological field stations. Yet, across all of these variations, there are particular practices that are integral to the core work of science, which center around the development of evidence-based explanations of the way the natural world works (Giere 1988). This in turn involves the generalizable practices of development of hypotheses from theories or models and testing these against evidence derived from observation and experiment (Lehrer and Schauble 2006; Giere 1988). Modeling—i.e., the collective action of developing, testing and refining models—has been described as the core epistemic and

representational practice in the sciences (Nersessian 1992; Lehrer and Schauble 2006; National Research Council 2008).

Furthermore, in the field of science education, there is also a growing understanding that the act of modeling is also a design activity because it involves carefully selecting aspects of the phenomenon to be modeled, identifying relevant variables, developing formal representations, and verifying and validating these representations with the putative phenomenon (Penner et al. 1998; Lehrer and Schauble 2006; Sengupta and Farris 2012). Modeling, therefore, requires articulating and instantiating appropriate objects and relations in a dialectical manner based on repeated cycles of designing mathematical or computational abstractions, making iterative comparisons of the generated representations and explanations with observations of the target phenomenon, and generating progressively more sophisticated explanations of the phenomenon to be modeled. Therefore, developing a computational model of a physical phenomenon involves key aspects of computational thinking identified by Wing (2008): identifying appropriate abstractions (e.g., underlying mathematical rules or computational methods that govern the behavior of relevant entities or objects), and iteratively refining the model through debugging and validation with corresponding elements in the real world.

Table 1 provides an overview of the correspondences between abstractions in computational thinking using *agent-based modeling and programming*—the particular genre of modeling and programming that we have adopted in our work—and the corresponding practices that are central to scientific inquiry.

2.2 Pedagogical benefits of integrating CT with science curricula

We believe that integrating CT and scientific modeling can be beneficial in a number of important ways. These include:

A. *Lowering the learning threshold by reorganizing scientific and mathematical concepts around intuitive computational mechanisms*: Sherin (2001) and diSessa (2000) argued that particular forms of programming could enable novice learners to access and reason about their intuitions about the physical world. Redish and Wilson (1993) argued that computational representations enable us to introduce discrete and qualitative forms of the fundamental laws, which can be much simpler to explain, understand, and apply, compared to the continuous forms traditionally presented in equation-based instruction. Furthermore, studies also suggest that in the domains of physics and biology, rather than organizing scientific phenomena in terms of abstract mathematical principles, the phenomena can be organized in a more intuitive fashion around computational mechanisms and principles (Redish and Wilson 1993; Sengupta and Wilensky 2011; Wilensky and Reisman 2006).
B. *Programming and computational modeling as representations of core scientific practices*: Soloway (1993) argued that learning to program amounts to learning how to construct mechanisms and explanations. Therefore, the ability to build computational models by programming matches core scientific practices that include model building and verification, as we pointed out earlier in the paper.

C. *Developing pre-algebra concepts through graphing functions and linked representations*: Much like computational environments for science, environments like SimCalc (Kaput 1994; Hegedus and Kaput 2004) ESCOT (Roschelle et al. 1999), and E-slate (Kynigos 2001, 2007) enable students to learn algebra by creating their own dynamic representations of concepts such as rate and proportion in the form of dynamic visualizations (animations) of computational actors.
D. *Contextualized representations make it easier to learn programming*: When computational mechanisms are anchored in real-world problem contexts, programming and computational modeling become easier to learn. Hambrusch et al. (2009) found that introducing computer programming to undergraduate students who were non-CS majors, in the context of modeling phenomena in their respective domains (physics and chemistry) resulted in higher learning gains (in programming), as well as a higher level of engagement in the task domain.

2.3 Why agent-based computation?

In this work, we focus on a particular genre of computational programming and modeling: Agent-based modeling and computation. In the agent-based paradigm, the user programs the behaviors of one or more agents—i.e., computational actors—by using simple computational rules, which are then executed or simulated in steps over time to generate an evolving set of behaviors. Among the earliest and best-known agent-based programming languages is Logo (Papert 1980). Logo and Logo-derivatives such as Boxer (diSessa 1985; diSessa and Abelson 1986) have been widely used to support children's learning in math (Papert 1980) and science (diSessa et al. 1991a, b; Roschelle and Teasley 1994) through design-based learning activities. The core building blocks for learning activities in Logo are computational procedures, which facilitate simultaneous learning of concepts about the phenomena being modeled and computational concepts, such as procedure abstraction, iteration, and recursion (Papert 1980; Harel and Papert 1991). Multi-agent-based computational modeling (MABM) is an extension of agent-based modeling, in which users can control the behaviors of thousands of agents at the same time.

MABMs have been shown to be particularly effective for modeling complex, emergent phenomena, i.e., phenomena in which counter-intuitive, aggregate-level effects emerge from simple interactions between many individual agents (e.g., formation of a traffic jam—while individual cars move forward, the traffic jam moves backward—see Resnick 1994). It enables us to represent a complex process or phenomenon in terms of simpler elements and mechanisms, often at a different level compared to the level at which the putative phenomenon is observed (Resnick 1994). Pedagogically, this also creates an opportunity for the student to develop the general practice of *problem decomposition*—i.e., represent a complex process or phenomenon in terms of simpler elements and mechanisms—a practice that is central in scientific modeling (Nersessian 1992), as well as computer science and software engineering (Ho 2001; Cross 2004). In terms of our own work, agent-based models have been shown to be effective pedagogical tools for learning and modeling aggregate-level and emergent phenomena in the domains of physics and ecology. For example, several scholars have shown that students' pre-instructional intuitions

about motion, while typically discrete and event-based, can be productively leveraged through appropriate scaffolding to generate correct understandings and representations of motion as a process of continuous change (diSessa et al. 1991b; diSessa 2001, 2004; Ford 2003; Sherin et al. 1993).

Research shows that when students learn using agent-based models and simulations, they first use their intuitive knowledge at the agent level to manipulate and reason about the behaviors of individual agents. As they visualize and analyze the aggregate-level behaviors that are dynamically displayed in the agent-based simulation environment, students can gradually develop multi-level explanations by connecting their relevant agent-level intuitions with the emergent aggregate-level phenomena (Resnick 1994; Wilensky and Resnick 1999; Klopfer et al. 2005; Sengupta and Wilensky 2011; Blikstein and Wilensky 2009). These scholars have argued that in most science classrooms, aggregate-level formalisms are the norm for teaching scientific phenomena, such as the Lotka-Volterra differential equation to explain how populations of different species in a predator–prey ecosystem evolve over time (Wilensky and Reisman 2006). In contrast, when complex phenomena (e.g., microscopic processes of electrical conduction, and waste consuming bacteria in an ecosystem) are represented in the form of multi-agent based models, elementary and middle school students (e.g., 4th and 5th graders) can access and understand those phenomena (Sengupta and Wilensky 2011; Dickes and Sengupta 2012; Tan and Biswas 2007). Such pedagogical approaches are constructivist in nature (Smith et al. 1993), as they enable children to build upon, rather than discard their repertoire of intuitive knowledge.

2.4 Why visual programming?

We focus on visual programming as the *mode* of programming and computational modeling to make it easier for middle school students to translate their intuitive knowledge of scientific phenomena (whether correct or incorrect) into executable models that they can then analyze by simulation. In visual programming environments, students construct programs using graphical objects, typically in a drag-and-drop interface (Kelleher and Pausch 2005; Hundhausen and Brown 2007). This significantly reduces students' challenges in learning the language syntax (compared to text-based programming), and thus makes programming more accessible to novices. This is an important affordance of visual programming, because prior research showed that students in a LOGO programming-based high school physics curriculum faced significant challenges in writing programs for modeling kinematics even after multiple weeks of programming instruction (Sherin et al. 1993). In the studies reported by Sherin et al. (1993) and diSessa et al. (1991a, b), middle and high school students required fifteen or more weeks of instruction, out of which, the first 5 weeks of classroom instruction were devoted solely to learning programming taught by a programming expert. Sherin et al. (1993) pointed out that, given the time constraints already faced by science (in their case, physics) teachers, the additional overhead associated with teaching students to program may simply prove prohibitive.

Some examples of agent-based visual programming environments are AgentSheets (Repenning 1993), StarLogo TNG (Klopfer et al. 2005), Scratch (Maloney et al. 2004), ToonTalk (Kahn 1996), Stagecast Creator (Smith et al. 2000), Kedama

(Oshima 2005) and Alice (Conway 1997). Users in all of all these environments can: (a) construct or design their programs by arranging icons or blocks that represent programming commands and (b) employ animations to represent the enactment (i.e., the execution) of the user-generated algorithm (i.e., program), albeit with varying degrees of algorithm visualization (Hundhausen and Brown 2007). However, it is important to note that these visual programming platforms have been typically been employed in K-12 classrooms with game design as the core learning activity. Our goal, in contrast, is to focus on designing agent-based visual programming languages specifically to support scientific modeling and simulation.

2.5 Selection of initial curricular topics

Our overarching, long-term goal is to support the development of computational thinking throughout the K-12 curriculum. As we highlighted earlier, this requires developing a long-term learning progression that spans multiple years. In this paper, we focus on only a small component of the learning progression: middle school science. Previous researchers who attempted to bring about integration between programming and science identified the following challenges: a) programming itself can be difficult to learn, and b) integrating programming with learning science can introduce challenges for students that pertain to learning programming but may not be relevant for understanding scientific concepts. We believe that addressing this issue requires re-thinking the design of programming languages, as well as, carefully selecting the curricular topics in science that leverages the affordances of the programming language.

The disciplines we have chosen as curricular contexts for science learning and modeling are kinematics (physics) and ecology (biology), which are common and important curricular topics at the K-12 level. Researchers have shown that K-12 students find these phenomena quite challenging to understand (Chi et al. 1994). Furthermore, it has been argued that students' difficulties in both the domains have similar epistemological origins, in that both kinematic phenomena (e.g., change of speed over time in an acceleration field) and system-level behaviors in an ecosystem (e.g., population dynamics) involve understanding aggregation of interactions over time (Reiner et al. 2000; Chi 2005). For example, physics educators have showed that understanding and representing motion as a process of continuous change has been shown to be challenging for novice learners (Halloun and Hestenes 1985; Elby 2000; Larkin et al. 1980; Leinhardt et al. 1990; McCloskey 1983). Novices tend to describe or explain any speed change(s) in terms of differences or relative size of the change(s), rather than describing speeding up or slowing down as a continuous process (Dykstra and Sweet 2009). Similarly, in the domain of ecology, biology educators have shown that while students have intuitive understandings of individual-level actions and behaviors, they find aggregate-level patterns that involve continuous dynamic processes—such as interdependence between species and population dynamics—challenging to understand without pedagogical support (Chi et al. 1994; Jacobson and Wilensky 2006; Wilensky and Novak 2010; Dickes and Sengupta 2012).

As discussed in Section 2.3, agent-based modeling is ideally suited for representing such phenomena, as it enables the learner to recruit their intuitions about agent-

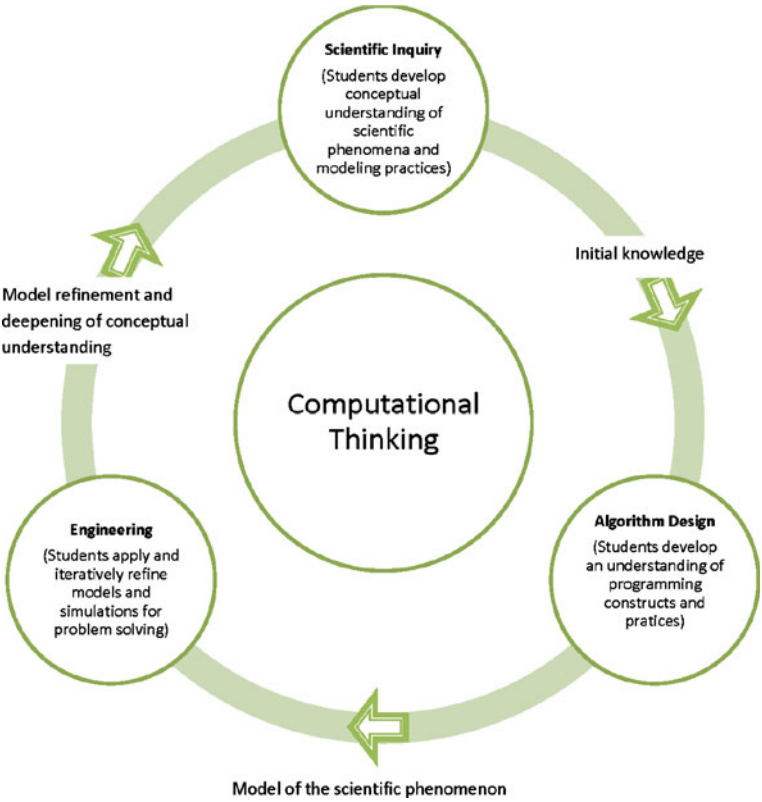

Fig. 1 CTSiM pedagogical framework for integrating computational thinking with K12 science

level behaviors and organize them through design-based learning activities (Kolodner et al. 2003), in order to explain aggregate-level outcomes. Studies have shown that pedagogical approaches based on agent-based models and modeling can allow novice learners to develop a deep understanding of dynamic, aggregate-level phenomena—both in kinematics and ecological systems by bootstrapping, rather than discarding their agent-level intuitions (Dickes and Sengupta 2012; Wilensky and Reisman 2006; Levy and Wilensky 2008; Sengupta et al. 2012).

3 Principles for system design

The conceptual framework for our pedagogical approach is illustrated in Fig. 1. The framework represents a typical sequence of *learning-by-design* activities that interweave action and reflection through engaging students in software design activities. Students begin with an initial understanding of the putative phenomenon, after which they design a model of the entities and processes involved in the phenomenon using an agent-based, visual programming platform. Students then iteratively simulate and refine the behavior of the model, by comparing their model to an "expert" model of the phenomenon, thereby developing explanations and arguments to deepen their understanding. Finally, students apply the developed model and the learned science

concepts in a new context for problem solving. Below, we outline our key design principles that guided our system design in order to support such a pedagogical approach. Some of these principles have been adapted from Sengupta (2011), and Sengupta and Farris (2012).

Support low-threshold and high-ceiling learning activities For a computing medium to be widely usable in K-12 science classrooms it must have the following affordances: (1) it should be easy to use for novice students; (2) learning activities should be well-integrated with the existing science curricula; (3) teachers, who are likely to have little or no computing background, should be able to master the system with minimal or no professional development; and (4) while *low-threshold* is a definite requirement for our system, it should not impose arbitrary ceilings on the scope and levels of complexity for modeling and analysis by students over time. Therefore, our goal here is to find a sweet spot that encapsulates *low-threshold* (i.e., easy to program), *wide walls* (i.e., students should be able to design a wide range of artifacts, such as animations, games, and science experiments, like Scratch™), and *high ceiling* (e.g., NetLogo (Wilensky 1999) supports advanced programming and modeling of complex phenomena using a library of intuitive programming primitives). While many of these design principles are shared by other modeling platforms (e.g., Logo, NetLogo, and Scratch), CTSiM is being designed specifically for pedagogical use in science classrooms, keeping *both teachers and students* in mind as users. This is reflected in the design of domain-specific programming primitives, scaffolds to make algorithms "live" and to support experimentation, and an explicit focus on curricular integration. We discuss these in detail in the rest of this section.

Incorporate multiple "liveness" factors as support for programming and learning by design Our challenge is to design a system that lets students seamlessly progress through cycles of construction, execution, analysis, reflection, and refinement using timely feedback from the simulation environment and scaffolding provided by the system. To support learning activities that involve rapid prototyping, our CTSiM system will offer learners a range of "liveness" factors for algorithm construction, visualization, analysis, and reflection (Tanimoto 1990). This primary design objective enables the learners to identify the relationship between their programs (i.e., algorithms) in the Construction World (see Section 4.1), and the resultant enactment of their simulations in the Enactment World (see Section 4.2). This involves developing scaffolds for supporting algorithm visualization (e.g., highlighting step-by-step execution of commands; controlling the delay between execution of successive commands). The goal is to provide timely feedback to students and avoid situations where lack of feedback and scaffolding may cause errors to accumulate late into the construction process, which makes the source of errors harder to detect. Accumulation of errors often leads to students being overwhelmed in terms of their cognitive abilities, resulting in their applying *trial and error* rather than *systematic* methods to conduct scientific inquiry (Segedy et al., to appear, 2012).

Verification and validation to support learning of expert models True scientific expertise involves understanding how knowledge is generated, justified, and evaluated by scientists and how to use such knowledge to engage in inquiry (Driver et al. 2000;

Duschl and Osborne 2002). Novice science learners engaging in these aspects of inquiry often struggle without extensive scaffolding (e.g., Klahr et al. 1990; Schauble et al. 1991; Sandoval and Millwood 2005). This is a challenge in computational modeling environments, where learning is often assumed to arise as students build their models and then verify them by comparing the behaviors generated against real data or behaviors of reference models representing the actual phenomena (Bravo et al. 2006). Therefore, building appropriate scaffolding and feedback is vital to the success of such environments. In CTSiM, learners can iteratively refine their programs by effectively comparing results of their simulation to an "expert" (i.e., canonically correct) simulation, understand the differences, and then map elements and constructs in their model to behaviors exhibited by their simulation and vice-versa.

Constructivist sequence of learning activities The sequence of learning activities in each domain is discussed in a later section. Our rationale behind the design of this sequence is grounded in the constructivist pedagogical approach that expert-like scientific understanding can develop by building upon and refining existing intuitive knowledge (Sengupta 2011; Dickes and Sengupta 2012; diSessa 1993; Hammer 1996; Smith et al. 1993; Sengupta and Wilensky 2009, 2011). For example, the initial learning activities leverage a naive conceptualization of the domains, and progressively scaffold them towards refinement. In kinematics, learners begin by inventing representations of motion in terms of measures of speed (how fast an object is moving) and inertia (innate tendency of an object to continue its current state of rest or motion, which often takes an anthropomorphic form in novice reasoning). They then gradually move to a force-based, more canonical description of motion in subsequent activities. In ecology, students begin with programming the behavior of single agents in the ecosystem (e.g., fish and duckweed in a fish tank) and gradually develop more complex programs for modeling the behavior and interaction of multiple species within the ecosystem (e.g., fish, duckweed, macro invertebrates, and species of bacteria).

4 Implementing CTSiM: Architecture and modules

We have implemented the design principles described above within the CTSiM learning environment. CTSiM includes three primary interface modules:

1. *The Construction World*: This module provides the visual programming interface for the student to build a computational model by composing program structures. Students select primitives from a library of commands and arrange them spatially, using a drag-and-drop interface to generate their programs. These programs define agent behaviors and interactions among different agents.
2. *The Enactment World*: This module is a microworld (Papert 1980; White and Frederiksen 1990) where the behaviors of agents defined in the Construction World can be visualized in a simulation environment. In our current implementation of CTSiM, user-defined models are implemented in the form of models in the Netlogo simulation environment (Wilensky 1999).
3. *The Envisionment World*: This module works closely with the enactment world, helping the student to set up experiments and analyze the behavior of their

models. In addition, students have access to the results generated by an expert simulation model, which run in lock step with the student-generated model. Students can use the envisionment world to compare the behaviors generated by their models against those generated by the expert model. Much of the scaffolding and feedback to help students refine, understand, and verify their models is also provided in this module.

The overall system architecture is illustrated in Fig. 2. Underlying the three interfaces discussed above are the model translator and model executor modules. These modules translate the student's visual programming code into equivalent NetLogo constructs so that the simulation can be executed in the NetLogo environment. An accompanying trace runner module helps the students align the visual behaviors generated by NetLogo to their visual programming constructs. The rest of this section provides a brief description of the three modules, and illustrates their functionality using a curricular unit in biology—a simple fish tank ecosystem that includes fish, duckweed, and bacteria that break down organic waste.

4.1 Construction world

The construction world allows students to build the relevant science model using an agent-based framework with relevant computational constructs. Figure 4 shows the drag-and-drop modeling interface in which students define their computational models for a simplified fish tank microworld. The students define the model for each type of agent by arranging and parameterizing a set of visual primitives, as illustrated for a partial fish agent model.

The visual primitives are named and iconically depicted in terms of their scientific function. These primitives are of three types: agent actions in the microworld (e.g., moving, eating, reproducing), sensing (e.g., vision, color, touch, toxicity), and controls for regulating the flow of execution in the computational model (e.g., conditionals, loops). Each visual primitive, in turn, is defined in terms of underlying computational primitives (with appropriate constraints and parameters), as illustrated in Fig. 3. The computational primitives provide a domain-independent set of computational constructs in a variety of categories: 1) changing (e.g., increasing or decreasing) the value of a property of the agent, 2) sensing conditions of the agent or its environment (e.g., reading local or global variable values), 3) creating or destroying agents (e.g., to model birth and death), and 4) conditionals and logical operators for controlling execution flow.

4.2 Enactment world

The enactment world interface allows the student to define a scenario (by assigning initial values to a set of parameters) and visualize the multi-agent-based simulation driven by their model. The CTSiM environment, implemented in Java, includes an embedded instance of NetLogo to implement the visualization and mechanics of the simulation. As the student builds a model, it is represented in the system as a code graph of parameterized computational primitives, illustrated in Fig. 3. This code graph remains hidden from the end-user (the learner), who accesses the

environment using the UI as shown in Figs. 4 and 5. The code graph enables the system to simulate the user-generated model for a given scenario by stepping through the graph and executing the computational primitives. The execution of the computational primitives indicated by the current (user) model drives the NetLogo simulation, as illustrated in Fig. 2. NetLogo visualization and plotting/measurement

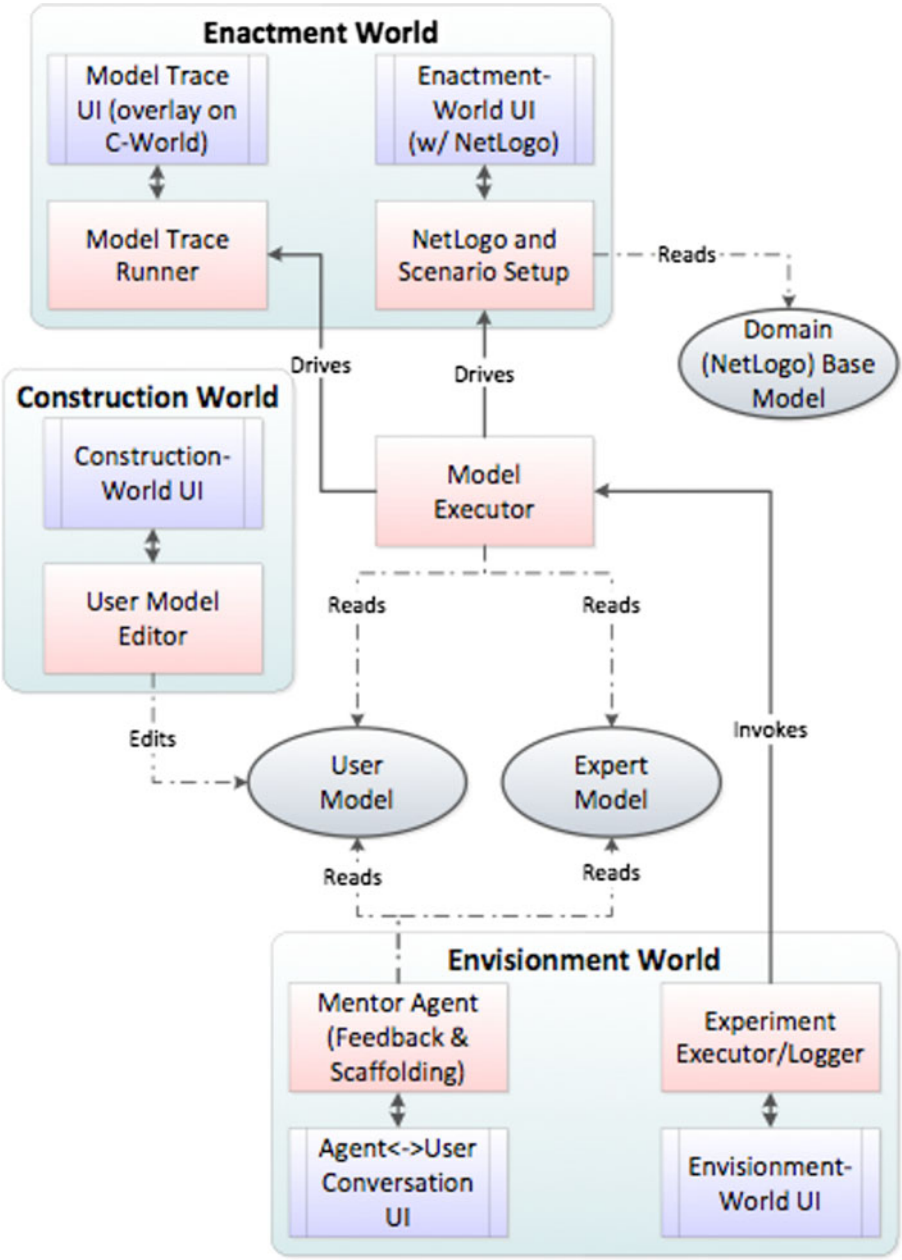

Fig. 2 CTSiM architecture

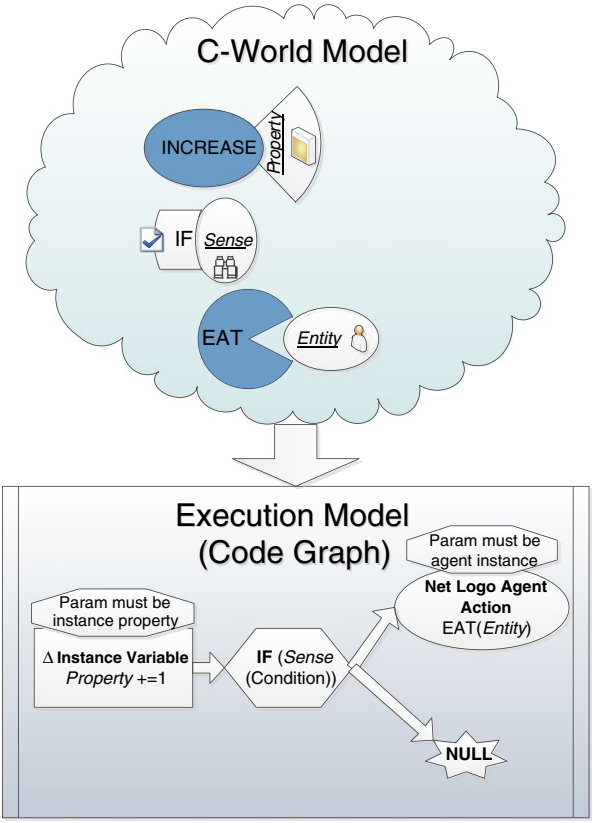

Fig. 3 Execution model for Construction-world

functionality (illustrated in Fig. 5) provide the students with a dynamic, real-time display of how their agents operate in the microworld, thus making explicit the emergence of aggregate system behaviors (e.g., from graphs of the population of a species over time).

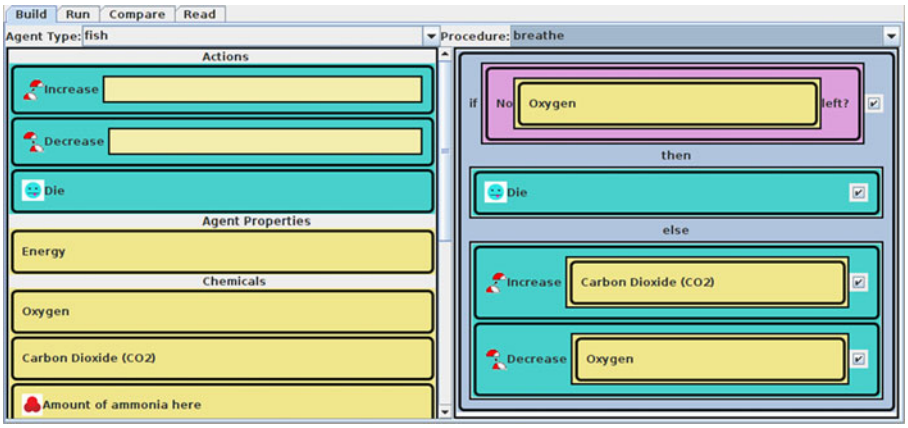

Fig. 4 The Ecology unit Construction world with a 'breathe' procedure for 'fish' agents

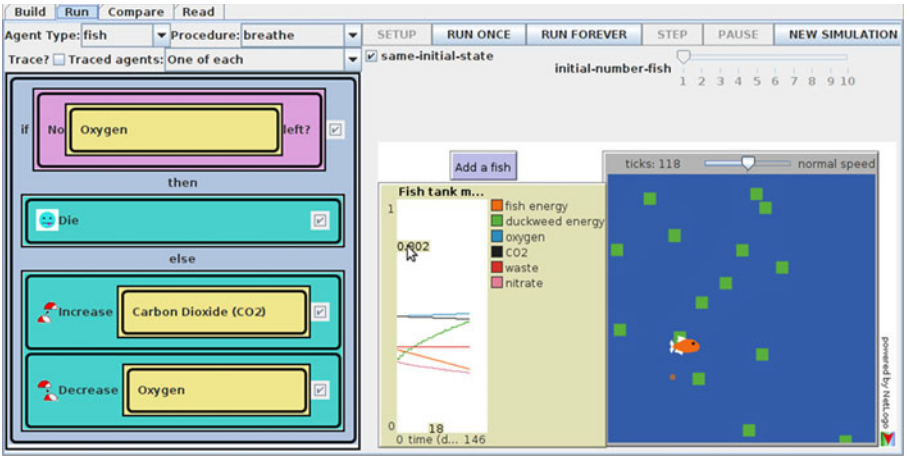

Fig. 5 A screenshot of the Enactment world for the Ecology macro-unit

Furthermore, since there is a one-to-one correspondence between the visual primitives in the construction world model and the computational primitives in the execution model (code graph), the system can highlight each primitive in the construction world as it is being executed in the enactment world. This model-tracing functionality will be leveraged to provide important scaffolding that supports model refinement and debugging activities.

4.3 Envisionment world

The envisionment world provides students with a space where they can systematically design experiments to test their constructed models and compare their model behaviors against behaviors generated by an "expert" model. A typical experimental setup would involve assigning values to agent parameters and variables defined in the student model, and simultaneously simulating the expert model with the same parameter values. Although the expert model is hidden, students may observe its behavior in comparison to their own model with side-by-side plots and microworld visualizations. With additional scaffolding and support, this allows students to make decisions on what components of their models they need to investigate, develop further, or check for errors to correct.

Work by Basu et al. (in review) details our approach to simulation investigation and discusses the scaffolds that we have developed to help students study and understand underlying model concepts and relations in a desert ecosystem microworld. With proper support and scaffolding, we believe that the overall process of model construction, analysis, comparison, and refinement will help students gain a better understanding of science (e.g., science phenomena and the scientific reasoning process), mathematics (e.g., measurement, graphs, and equations), and computational constructs and methods. Therefore, the Envisionment world provides students the opportunity to conduct systematic experiments that compare their models' behavior against behavior generated by an "expert" model. This side-by-side comparison of plots and microworld visualizations for the two models makes it easier for students to investigate and revise their models.

For example, each cycle of modeling will begin with students critiquing the fit between the current version of their model, and the target agent- and aggregate-level behaviors generated from the expert model. If differences in system behavior (between the expert model and the student's model) are identified through the critique phase, the student will then be scaffolded in creating an explanation/claim about the mechanisms in their constructed model that could lead to the observed differences. The methods discussed in the construction and enactment worlds will support this interaction and help students identify the segments of their model that are contributing to differences in system behavior. The goal is to scaffold explanation about the underlying causal mechanism that simultaneously identifies an issue and proposes corrective action.

After creating an explanation about the source of differences between the agent-level program and aggregate-level outputs, the student will be scaffolded to identify and collect further evidence for, or against, the proposed causal mechanism. This can include evidence collected during the critiquing phase or during subsequent experimentation. Research has shown that students initially tend to focus only on evidence that supports their claims and ignore the evidence that contradicts their claims, in a manner very similar to the ways in which scientists have historically dealt with anomalous data (Chinn and Brewer 1993). Therefore, the scaffolding should help students search for evidence that might contradict their claims, as well as evidence that supports their claims.

4.4 Model execution and model tracing

In a CTSiM unit, the expert model is described using the same set of visual primitives available to the students. Further, each visual primitive is defined by one or more (parameterized) computational primitives, as described in Section 4.1. This allows both student-built and pre-defined expert models to be executed, analyzed, and compared in the same, well-defined computational language (i.e., as code graphs like the one illustrated in Fig. 3). In CTSiM the model executor component, illustrated in Fig. 2, can translate a (student-built or expert) model's code graph into corresponding NetLogo code, which is then combined with the domain base model. The base model provides the supporting NetLogo code for visualization and other housekeeping aspects of the simulation that are not directly relevant to the learning goals of the unit. The combined model forms a complete, executable NetLogo simulation, which can be run in the Enactment or Envisionment Worlds.

Model tracing, or providing supports for making algorithms "live" (Tanimoto 1990), involves highlighting the part of the code (visual programming commands) that is being currently executed, so that the student can better understand how their generated code affects the behavior of the agent(s) in the simulation. This visual tracing can be particularly important for helping students better understand the correspondence between their models and simulations, as well as for identifying and correcting model errors. The execution of the translated model retains the execution speed of a NetLogo simulation. But in order retain this speed, model executor does not directly provide the students any support for model tracing. Instead, it provides an alternate translation and execution path, indicted as "model trace runner" in Fig. 2. Rather than translating the entire student-generated model into

NetLogo code, using this functionality, each visual primitive is translated separately. The executor then steps through the model, calling the individual chunks of NetLogo code corresponding to the execution of each visual primitive. During this step-by-step execution, the model tracer provides visual highlights on the student's model corresponding to the visual programming primitive that was just executed.

5 Curricular modules & activities

We have developed two curricular modules in the CTSiM environment (Basu et al. 2012): (1) a kinematics unit that focuses on modeling Newtonian mechanics phenomena such as the trajectory, velocity, and acceleration of balls placed on different inclined planes, as well as using mathematically meaningful aesthetic representations (e.g., geometric shapes) to represent various classes of kinematic phenomena (e.g., constant speed, constant acceleration) (Sengupta 2011; Sengupta et al. 2012); and (2) an ecology unit (Tan and Biswas 2007) that emulates a simplified fish tank environment, which includes fish, duckweed, and bacteria for breaking down organic waste; it primarily models a simplified food chain and the waste cycle focusing on how organic waste is broken down in steps by bacteria to produce nitrates, which provide nutrition for duckweed. The focus of this unit is on the study of interdependence and balance in ecosystems.

In terms of learning programming, these modeling activities introduce students to fundamental programming constructs (e.g., agents, conditionals, loops, variables, etc.). For example, the first two modeling tasks in both kinematics and ecology involve students using three main types of constructs: agents (various species of organisms in ecology; physical objects in kinematics), conditionals (indicating need-based interactions between agents in ecology, and effect of different conditions, e.g., terrains, in kinematics), and loops (for repetitions of an action(s) based on the current value of an agent's or environmental property/variable). In subsequent modeling activities, students learn to define variables (e.g., defining attributes of agents and breeds of agents), as well as code-reuse and encapsulation (e.g., using portions of existing code to model behavior of new agents). In later, more complex, modules, students will explore class hierarchy/inheritance connected to science through relation to taxonomy and class/type properties and behaviors, as well as the nature of physical "laws" that apply to all objects (e.g., in kinematics).

Note that our rationale behind sequencing the two domains in the curriculum was guided by the programming complexities involved in modeling phenomena in the two domains. For example, while the kinematics learning activities described below required the students to program the behavior of a single computational agent, modeling the fish tank ecosystem would require the students to program the behaviors of and interactions between multiple agents. This necessitated that we introduce students to single-agent programming before introducing them to multi-agent programming—therefore, in our curricular sequence, students learn kinematics first, and then ecology.

From the perspective of developing mathematical expertise, students in both curricular units focus on: (1) generating graphs of aggregations (e.g., averages) over time, (2) identifying and developing basic mathematical relationships between variables and statistical properties of populations evident from graphs (e.g., linear vs.

quadratic relationships, averages, range, variance/standard-deviation), and (3) understanding rates through designing multiple, linked representations of rates and other time-based/time-variant relationships.

5.1 Kinematics unit

For the kinematics unit, our central learning objective was for students to be able to represent and understand motion as a process of continuous change. Following Sengupta et al. (2012), activities in the Kinematics unit were divided into three phases, as discussed below:

Phase 1: Turtle Graphics for Constant Speed and Constant Acceleration—We introduced students to programming commands by showing them how to manipulate different elements in the user interface. Then, we asked them to generate algorithms to draw simple shapes (squares, triangles and circles) to familiarize them with programming primitives like "forward", "right turn", "left turn", "pen down", "pen up" and "repeat". Next, we asked students to modify their algorithms and generate spiraling shapes in which each line segment is longer (or shorter) than the previous one. This exercise introduced students to the "speed-up" and "slow-down" commands, and it gave them a chance to explore the relationship between speed, acceleration, and distance.

Phase II: Conceptualizing and re-representing a speed-time graph—In this activity, students generated shapes such that the length of segments in the shapes were proportional to the speed in a given speed-time graph. Figure 6 depicts the speed-time graph provided to all students, along with a sample student output where the initial spurt of acceleration is represented by a small growing triangular spiral, the gradual deceleration by a large shrinking square spiral, and constant speed by a triangle. The focus was on developing mathematical measures from meaningful estimation and mechanistic interpretations of the graph, and thereby gaining a deeper understanding of concepts like speed and acceleration.

Phase III: Modeling motion of an agent to match behavior of an expert model—For this activity, students modeled the behavior of a roller coaster as it moved on

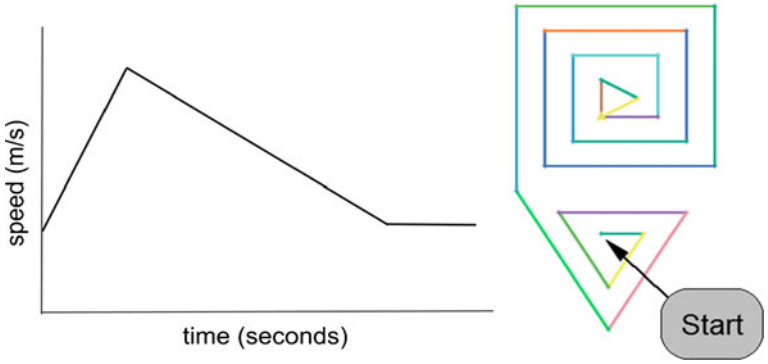

Fig. 6 Acceleration represented in a speed-time graph (*left*), and in the CTSiM Enactment world (*right*)

different segments of a track: up (pulled by a motor), down, flat, and then up again. Students were first shown the simulation results produced by an 'expert' roller coaster model in the Envisionment world. Then, they were asked to conceptualize and build their own agent model to match the observed expert roller coaster behavior for all of the segments.

5.2 Ecology unit

For the Ecology unit, students modeled a closed fish tank system in two steps: (1) a macro-level semi-stable model of the behavior of fish and duckweed; and (2) a micro-level model of the waste cycle with bacteria. The macro model included several key relations: (1) the food chain and respiration cycles of the fish and duckweed, (2) the macro-level elements of the waste cycle (fish produce waste, duckweed consume nitrates), and (3) the reproduction of duckweed. The non-sustainability of the macro-level model—i.e., the fish and the duckweed gradually dying off—creates a context for reflection about the underlying reasons for the population decay. This in turn provided the transition to the micro model. When prompted to think about why the system was not self-sustaining, previous studies showed that students can indeed identify the continuously increasing fish waste as the main cause. The micro model provides students with an opportunity to further deepen their understanding of the process of waste cycle by introducing the role of bacteria in the system.

In the micro-model, the students modeled the waste cycle and the designed computational relationships between chemical processes in order to simulate conversion of toxic ammonia in the fish waste to nitrites, and then nitrates, which sustained the duckweed. The graphs generated from the expert simulation helped students understand the producer-consumer relations: (1) *Nitrosomonas* bacteria consume ammonia and produces nitrites; (2) nitrites are consumed by *Nitrobacter* bacteria to produce nitrates, which provide food for the duckweed.

6 Pilot study

6.1 Setting and method

In order to assess the effectiveness of our pedagogical approach, we conducted a study with 6th-grade students ($n=24$) from an ethnically diverse middle school in central Tennessee in the United States. Fifteen students worked on the system in the school library with one-on-one guidance from members of our research team (Scaffolded or S-Group), while the remaining nine students in the class worked on the system in the classroom (Classroom or C-Group). In the S-Group, students worked one-on-one with a researcher and were provided verbal scaffolds. In the C-Group, students received minimal one-on-one scaffolding, as they had one classroom teacher in charge of the instructional activities. The C group was taught in a lecture format by one of the researchers and the class teacher who introduced them to each activity, after which the students had to conduct each activity individually. The students in the C-group were also provided assistance from the researchers if they raised their hand and asked for help.

Students first interacted with the Kinematics unit in hour-long sessions for 3 days, after which they interacted with the Ecology unit in hour-long sessions for another 3 days. After completing the ecology micro model, the S group received an additional scaffold: they discussed the combined micro–macro model with their assigned researcher and were shown how the two models were causally linked to support sustainability. Students were given the paper-and-pencil task of building a causal model of the cycles, and then prompted to use this representation to explain the effects of removing one agent on the stability of the cycle.

On day 1 of the study, we administered pre-tests for both units. Students worked on the kinematics unit from days 2 to 4, and then took the kinematics post-test on day 5. This was followed by work on the ecology unit from days 6 to 8, and the ecology post-test on day 9.

Our goal was to investigate whether students, after interacting with the CTSiM curricular units, were able to show gains in the relevant curricular learning goals. The Kinematics pre/post-tests were designed to assess students' reasoning using mathematical representations of motion (e.g., graphs). Our goal was to assess whether agent-based modeling improved their abilities to both generate and explain these representations. Specifically, students were asked interpret and explain speed versus time graphs, and to generate diagrammatic representations to explain motion in a constant acceleration field. For the Ecology unit, the pre- and post-tests focused on students' understanding of roles of species in the ecosystem, interdependence among the species, the waste and respiration cycles, and how a specific change in one species affected the others. Some of the questions required students to use declarative knowledge about the fish tank system (we coded these items as Declarative Knowledge Check or DKC); other questions assessed Causal Reasoning about entities using the Declarative Knowledge (coded as CRDK); and a Transfer Question (TQ) required students to reason about the carbon cycle.

6.2 Findings: learning gains in kinematics and ecology

The pre-test scores and the mean TCAP (Tennessee Comprehensive Assessment Program) science scores suggested differences in prior knowledge and abilities of the S and C groups (t=3.15, p<0.005) in their mean TCAP science. Hence we computed repeated measures ANCOVA with TCAP science scores as a covariate of the pre-test scores to study the interaction between time and condition. There was a significant effect of condition (i.e., S-Group versus C-Group) on pre-post learning gains in ecology ($F(1,21)=37.012$, $p<0.001$), and a similar trend was seen in kinematics ($F(1,21)=4.101$, $p<0.06$). The plots in Fig. 7 show that the S group's adjusted gains were higher than the C group in both units.

We conducted paired sample t-tests to compare pre-to-post gains for each group. These scores represent the mean total score (i.e., total score averaged over all students), where the total score for each student was calculated by counting the number of correct responses for all the questions in Kinematics and in Ecology. Table 2 shows that the intervention produced statistically significant gains for ecology unit ($p<0.001$ for S-group; $p<0.01$ for C-Group), but for the kinematics unit, the gains were statistically less significant for the S-Group ($p<0.1$), and not significant for the C-Group.

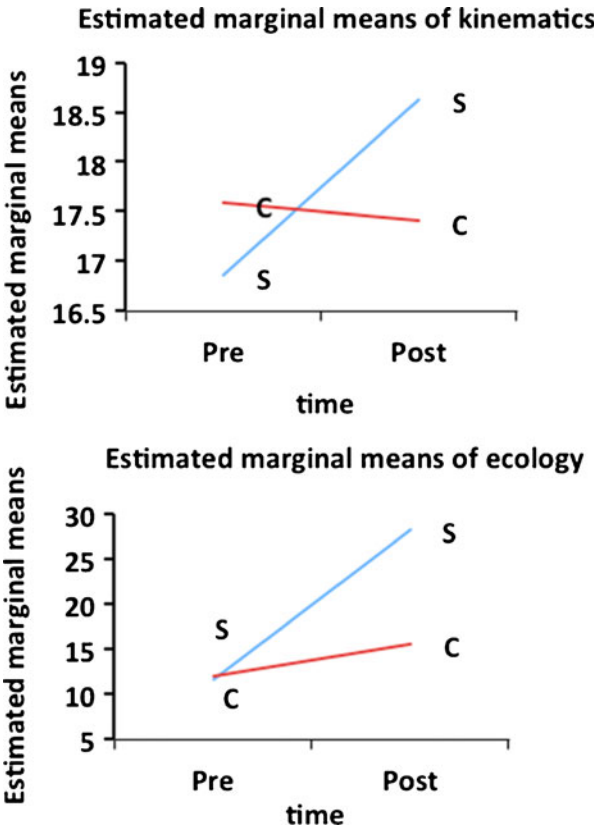

Fig. 7 Comparison of gains between groups using TCAP scores as a covariate

The reduction in statistical significance for the kinematics unit may be attributed to a *ceiling effect* in the students' pre-test scores, given that students entered the instructional setting with a significantly higher score in the kinematics pre-test than the ecology pre-test. However, it is noteworthy that for both the kinematics and the ecology units, the S group, which received direct one-on-one scaffolding, showed higher learning gains than the C group.

Table 2 Paired *t*-test results for Kinematics and Ecology pre and post test scores

	Kinematics				Ecology			
	PRE (S.D.) (max=24)	POST (S.D.) (max=24)	*t*-value	*P*-value (2-tailed)	PRE (S.D.) (max=35.5)	POST (S.D.) (max=35.5)	*t*-value	*P*-value (2-tailed)
S-Group (n=15)	18.07 (2.05)	19.6 (2.29)	2.699	0.017	13.03(5.35)	29.4(4.99)	8.664	<0.001
C-Group (n=9)	15.56 (4.1)	15.78 (4.41)	0.512	0.622	9.61(3.14)	13.78(4.37)	3.402	<0.01

Of note are students' responses for a particular question in the Kinematics pre- and post-tests in which they were asked to diagrammatically represent the temporal trajectory of a ball dropped from the same height on the earth, and on the moon. The students were also asked to explain their drawings and generate graphs of speed versus time for the two scenarios. The S group showed significant gains ($p<0.0001$) on this question, while the C group showed an increasing trend, although it was not significant ($p=0.16$). This suggests that students, after interacting with CTSiM, were able to represent and interpret motion as a process of continuous change in position and speed. Given that one of our central learning goals involved being able to represent motion as a continuous process of change, we believe that this result is noteworthy as it provides evidence for the effectiveness of our pedagogical approach.

For the ecology unit, the S-Group students showed gains for all the questions, although not all the gains were statistically significant. Table 3 reports normalized learning gains (i.e., gain/maximum possible gain) by question category for both the groups. Significant gains were observed on the DKC and CRDK questions, which can be attributed to an increased awareness of the entities in the fish tank and their relations with other species. For example, pre-test results indicated that the students did not initially know about the bacteria and their roles. Though students in both groups were told about the role of bacteria during the intervention, the supplementary causal-reasoning activity helped the S-group students gain a better understanding of the interdependence among the species. The S group's gains on the TQ were not significant due to a ceiling effect (most students had strong prior knowledge about the carbon cycle). On the contrary, the C-Group gained only on the CRDK questions, though less than the S-Group ($F(1,21)=21.06$, $p<0.001$). This can be explained by the C group's minimal scaffolding and, especially, the absence of scaffolds targeted towards causal reasoning.

7 Discussion & conclusions

7.1 A framework for integrating computational thinking and science education

In general, computational thinking involves being able to reason at multiple levels of abstraction (Kramer 2007), mathematical reasoning, and design-based thinking (Wing 2008). It also involves as well as using these kinds of reasoning for contextual problem solving (Guzdial 2008; Wing 2008). Our paper proposes a theoretical framework through which these aspects of CT can be integrated with curricular

Table 3 Normalized learning gains on categories of Ecology questions

S-Group normalized gains			C-Group normalized gains		
DKC	CRDK	TQ	DKC	CRDK	TQ
0.865 (<0.0001)	0.725 (<0.0001)	0.495 (0.11)	0 (NA)	0.192 (<0.01)	0 (NA)

modules in K-12 science classrooms via agent-based modeling. The framework we proposed accomplishes the following:

a) It elucidates the relationship between CT and scientific expertise—we highlight the pedagogical benefits of integrating CT with science learning, including their synergies;
b) It provides a justifications for the choice of a particular programming paradigm—agent-based computation—and a mode of programming—visual programming—in order to facilitate scientific modeling alongside CT;
c) It outlines the rationale behind choosing an initial set of topics (kinematics and ecology) in science that are amenable to our technology, but at the same time illustrate the breadth and generality of our approach;
d) It also proposes a set of design principles for integrating CT and Science Learning, based on which we designed the CTSiM learning environment. These include: a) supporting low-threshold as well as high-ceiling learning activities; b) design of programming primitives, c) supporting algorithm visualization; and d) sequencing learning activities in a constructivist fashion.

However, in addition to proposing a theoretical framework, we also provide an example of how one might implement such a framework in practice. To do so, we also presented the computational architecture of a learning environment—CTSiM—that is based on this theoretical framework, and includes a visual programming language and a modeling environment. Our goal in doing so our goal was to highlight the software components, including elements of the user-interface, which were needed in order to implement the design principles we identified in the theoretical framework. The three "worlds" in our system—Construction World, Enactment World and Envisionment World—extend the functionalities of the learning environment beyond programming, and provide explicit supports for developing and reasoning with scientific models. Each of these worlds serves specific pedagogical functions, and further, complements each other in terms of the pedagogical support and opportunities that each of them provides for the learners to support CT and STEM learning.

Our discussion of the curricular units also highlights the design rationale behind the sequencing of the learning activities, which in grounded in terms of the complexities involved in learning the relevant aspects of programming, as well as the need to support students' model construction in a gradual, constructivist fashion. Finally, we presented preliminary results from a pilot study, in which students interacted with these curricular activities in the CTSiM environment. These results provide preliminary evidence of the effectiveness of our proposed pedagogical approach. We discuss this in more detail below.

7.2 Interpretative summary of findings from the pilot study

Our results from the Pilot study indicate that the learning environment resulted in significant learning gains, as measured by the difference in students' pre- and post-test scores, for both the Kinematics and Ecology units. Furthermore, the difference in learning gains between the Scaffolded and Classroom groups indicate the necessity of

importance of one-on-one scaffolding to support student learning in a learning environment that integrates CT and science learning. We are currently in the process of analyzing and categorizing the types of scaffolds that was required by each student in the S-group. This is not only important in order to understand how students developed their understandings and representations as they interacted with CTSiM, but it also has important implications for system design. In future work, we will integrate such scaffolds into the CTSiM environment by building scaffolding tools and providing feedback via a virtual mentor agent.

7.3 Laying the foundation for a long-term learning progression

In this paper, we have laid the foundation for a longer-term learning progression that integrates CT with scientific modeling spanning several years. Central to this mission is flexibility across multiple scientific domains. Our framework and environment can engage students in agent-based modeling across these domains while employing the same basic programming constructs and modeling environment for all units. In doing so, our primary goal focuses on enabling students to discover and represent computational abstractions across multiple domains in terms of the underlying, generalizable computational/mathematical constructs and practices (e.g., control flow, variables, debugging), as well as the domain-general representational practices involved in modeling (e.g., problematizing, experimentation, generating inscriptions, verification, and validation). In designing this system, our goal is to integrate computational thinking with existing K-12 science curricula (as has been recommended by the ACM K-12 Taskforce (2003)), without necessitating the development of new science standards, and without introducing a programming course separate and disjoint from the science curricula. This also lays the groundwork for the development of a long-term curricular progression in which students can engage in learning science using computational modeling and thinking over a span of multiple years. This is consistent with the findings from research in developmental psychology and science education, which show that the development of scientific thinking, even when it builds on students' intuitive knowledge and competencies, requires multiple years of meaningful immersion in authentic learning experiences (Lehrer et al. 2008).

Acknowledgments Thanks to Amanda Dickes, Amy Voss Farris, Gokul Krishnan, Brian Sulcer, Jaymes Winger, and Mason Wright (in no particular order), who helped in developing the system and running the study. Earlier versions of the paper were presented at CSEDU 2012, and ICCE 2012. This work is partially supported by NSF IIS # 1124175 and NSF Early CAREER # 1150230.

References

ACM K-12 Taskforce. (2003). *A Model Curriculum for K-12 Computer Science: Final Report of the ACM K-12 Task Force Curriculum Committee*. New York, NY: CSTA.

Aristotle (350 BCE/2002) *Nichomachean ethics*. New York: Oxford University Press.

Basu, S., Sengupta, P., & Biswas, G. (In Review). A scaffolding framework to support learning in multi-agent based simulation environments. *Research in Science Education*.

Basu, S., Kinnebrew, J., Dickes, A., Farris, A. V., Sengupta, P., Winger, J., & Biswas, G. (2012). A Science Learning Environment using a Computational Thinking Approach. In: *Proceedings of the 20th International Conference on Computers in Education*, Singapore.

Blikstein, P., & Wilensky, U. (2009). An atom is known by the company it keeps: A constructionist learning environment for materials science using Agent-Based Modeling. *International Journal of Computers for Mathematical Learning, 14*, 81–119.

Bravo, C., van Joolingen, W. R., & deJong, T. (2006). Modeling and simulation in inquiry learning: Checking solutions and giving advice. *Simulation, 82*(11), 769–784.

Chi, M. T. H. (2005). Common sense conceptions of emergent processes: Why some misconceptions are robust. *Journal of the Learning Sciences, 14*, 161–199.

Chi, M. T. H., Slotta, J. D., & de Leeuw, N. (1994). From things to processes: A theory of conceptual change for learning science concepts. *Learning and Instruction, 4*, 27–43.

Chinn, C. A., & Brewer, W. F. (1993). The role of anomalous data in knowledge acquisition: A theoretical framework and implications for science instruction. *Review of Educational Research, 63*, 1–49.

Conway, M. (1997). *Alice: Easy to Learn 3D Scripting for Novices*, Technical Report, School of Engineering and Applied Sciences, University of Virginia, Charlottesville, VA.

Corcoran, T., Mosher, F., & Rogat, A. (2009). *Learning progressions in science: An evidence-based approach to reform (RR-63)*. Philadelphia, PA: Consortium for Policy Research in Education.

Cross, N. (2004). Expertise in design: An overview. *Design Studies, 25*, 427–441.

Dickes, A., & Sengupta, P. (2012). Learning Natural Selection in 4th Grade with Multi Agent-Based Computational Models. *Research in Science Education*. doi:10.1007/s11165-012-9293-2.

diSessa, A. A. (1985). A principled design for an integrated computational environment. *Human-Computer Interaction, 1*(1), 1–47.

diSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and Instruction, 10*(2/3), 105–225.

diSessa, A. A. (2000). *Changing minds: Computers, learning, and literacy*. Cambridge, MA: MIT Press.

diSessa, A. A. (2004). Metarepresentation: Native competence and targets for instruction. *Cognition and Instruction, 22*(3), 293–331.

diSessa, A. A. (2001). *Changing minds: Computers, learning, and literacy.* The MIT Press.

diSessa, A. A., & Abelson, H. (1986). BOXER: A reconstructible computational medium. *Communications of ACM, 29*(9), 859–868.

diSessa, A. A., Abelson, H., & Ploger, D. (1991a). An overview of boxer. *Journal of Mathematical Behavior, 10*(1), 3–15.

diSessa, A., Hammer, D., Sherin, B., & Kolpakowski, T. (1991b). Inventing graphing: Children's meta-representational expertise. *Journal of Mathematical Behavior, 10*(2), 117–160.

Driver, R., Newton, P., & Osborne, J. (2000). Establishing the norms of scientific argumentation in classrooms. *Science Education, 84*(3), 287–313.

Duschl, R. (2008). Science education in three part harmony: Balancing conceptual, epistemic and social learning goals. In J. Green, A. Luke, & G. Kelly (Eds.), *Review of research in education* (Vol. 32, pp. 268–291). Washington, DC: AERA.

Duschl, R. A., & Osborne, J. (2002). Supporting and promoting argumentation discourse in science education. *Studies in Science Education, 38*, 39–72.

Dykstra, D. I., Jr., & Sweet, D. R. (2009). Conceptual development about motion and force in elementary and middle school students. *American Journal of Physics, 77*(5), 468–476.

Edelson, D. C. (2001). Learning-for-use: A framework for the design of technology-supported inquiry activities. *Journal of Research in Science Teaching, 38*(3), 355–385.

Elby, A. (2000). What students' learning of representations tells us about constructivism. *Journal of Mathematical Behavior, 19*, 481–502.

Ford, M. J. (2003). Representing and meaning in history and in classrooms: Developing symbols and conceptual organizations of free-fall motion. *Science & Education, 12*(1), 1–25.

Giere, R. N. (1988). *Explaining science: A cognitive approach*. Chicago: University of Chicago Press.

Guzdial, M. (1995). Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments, 4*(1), 1–44.

Guzdial, M. (2008). *Paving the way for computational thinking*. Communications of the ACM: Education Column. 51(8).

Halloun, I. A., & Hestenes, D. (1985). The initial knowledge state of college physics students. *American Journal of Physics, 53*(11), 1043–1056.

Hambrusch, S., Hoffmann, C., Korb, J. T., Haugan, M., & Hosking, A. L. (2009). A multidisciplinary approach towards computational thinking for science majors. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (SIGCSE '09). ACM, New York, NY, USA, 183–187.

Hammer, D. (1996). Misconceptions or p-prims: How may alternative perspectives of cognitive structure influence instructional perceptions and intentions? *Journal of the Learning Sciences, 5*(2), 97–127.

Harel, I., & Papert, S. (1991). Software design as a learning environment. *Constructionism.* Norwood, NJ: Ablex Publishing Corporation. pp. 51–52. ISBN 0-89391-785-0.

Hegedus, S. J., & Kaput, J. J. (2004). An Introduction to the Profound Potential of Connected Algebra Activities: Issues of Representation, Engagement, and Pedagogy. *Proceedings of the 28th Conference of the International Group for the Psychology of Mathematics Education, 3*, 129–136.

Ho, C. H. (2001). Some phenomena of problem decomposition strategy for design thinking: Differences between novices and experts. *Design Studies, 22*(1), 27–45.

Hundhausen, C. D., & Brown, J. L. (2007). What You See Is What You Code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages and Computing, 18*, 22–47.

Jacobson, M., & Wilensky, U. (2006). Complex systems in education: Scientific and educational importance and implications for the learning sciences. *Journal of the Learning Sciences, 15*(1), 11–34.

Kahn, K. (1996). ToonTalk: An animated programming environment for children. *Journal of Visual Languages and Computing*.

Kaput, J. (1994). Democratizing access to calculus: New routes using old routes. In A. Schoenfeld (Ed.), *Mathematical thinking and problem solving* (pp. 77–156). Hillsdale, NJ: Lawrence Erlbaum.

Kelleher, C., & Pausch, R. (2005) Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys,* Vol. (37) 83–137.

Klahr, D., Dunbar, K., & Fay, A. L. (1990). Designing good experiments to test bad hypotheses. In J. Shrager & P. Langley (Eds.), *Computational models of scientific discovery and theory formation* (pp. 355–401). San Mateo, CA: Morgan Kaufman.

Klopfer, E., Yoon, S., & Um, T. (2005). Teaching complex dynamic systems to young students with StarLogo. *The Journal of Computers in Mathematics and Science Teaching, 24*(2), 157–178.

Kolodner, J. L., Camp, P. J., Crismond, D., Fasse, B., Gray, J., Holbrook, J., Puntambekar, S., & Ryan, M. (2003). Problem-based learning meets case-based reasoning in the middle-school science classroom: Putting learning by design into practice. *The Journal of Learning Sciences, 12*(4), 495–547.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM, 50*(4), 36–42. April 2007.

Kynigos, C. (2001). *E-slate Logo as a basis for constructing microworlds with mathematics teachers* (pp. 65–74). Lintz, Austria: Proceedings of the Ninth Eurologo Conference.

Kynigos, C. (2007). Using half-baked microworlds to challenge teacher educators' knowing. *Journal of Computers for Math Learning, 12*(2), 87–111.

Larkin, J. H., McDermott, J., Simon, D. P., & Simon, H. A. (1980). Expert and novice performance in solving physics problems. *Science, 208*, 1335–1342.

Lehrer, R., & Schauble, L. (2006). Cultivating model-based reasoning in science education. In R. K. Sawyer (Ed.), *The Cambridge handbook of the learning sciences* (pp. 371–388). New York: Cambridge University Press.

Lehrer, R., Schauble, L., & Lucas, D. (2008). Supporting development of the epistemology of inquiry. *Cognitive Development, 23*(4), 512–529.

Leinhardt, G., Zaslavsky, O., & Stein, M. M. (1990). Functions, graphs, and graphing: Tasks, learning and teaching. *Review of Educational Research, 60*, 1–64.

Levy, S. T., & Wilensky, U. (2008). Inventing a "mid-level" to make ends meet: Reasoning through the levels of complexity. *Cognition and Instruction, 26*(1), 1–47.

Locke, J. (1690/1979). *An essay concerning human understanding*. New York: Oxford University Press.

Maloney, J., Burd, L., Kafai, Y., Rusk, N., Silverman, B., & Resnick, M. (2004) Scratch: A sneak preview. In *Proceedings of Creating, Connecting, and Collaborating through Computing,* 104–109.

McCloskey, M. (1983). Naive theories of motion. In D. Gentner & A. Stevens (Eds.), *Mental models* (pp. 299–324). Hillsdale, NJ: Lawrence Erlbaum.

National Research Council. (2008). *Taking science to school: Learning and teaching science in grades K–8*. Washington, DC: National Academy Press.

National Research Council. (2010). *Report of a workshop on the scope and nature of computational thinking*. Washington, DC: The National Academies Press.

Nersessian, N. J. (1992). How do scientists think? Capturing the dynamics of conceptual change in science. In R. N. Giere (Ed.), *Cognitive models of science* (pp. 3–45). MN: University of Minnesota Press. Minneapolis.

Oshima, Y. (2005). Kedama: A GUI-based interactive massively parallel particle programming system. *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05).*

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books, Inc.

Papert, S. (1991). Situating constructionism. In I. Harel & S. Papert (Eds.), *Constructionism*. Norwood, NJ: Ablex Publishing Corporation.

Penner, D. E., Lehrer, R., & Schauble, L. (1998). From physical models to biomechanics: A design-based modeling approach. *Journal of the Learning Sciences, 7*(3–4), 429–449.

Perkins, D. N., & Simmons, R. (1988). Patterns of misunderstanding: An integrative model for science, math, and programming. *Review of Educational Research, 58*(3), 303–326.

Redish, E. F., & Wilson, J. M. (1993). Student programming in the introductory physics course: M.U.P.P.E.T. *American Journal of Physics, 61*, 222–232.

Reiner, M., Slotta, J. D., Chi, M. T. H., & Resnick, L. B. (2000). Naive physics reasoning: A commitment to substance-based conceptions. *Cognition and Instruction, 18*(1), 1–34.

Repenning, A. (1993). Agentsheets: A tool for building domain-oriented visual programming. *Conference on Human Factors in Computing Systems*, 142–143.

Resnick, M. (1994). *Turtles, termites, and traffic jams: Explorations in massively parallel microworlds*. Cambridge, MA: MIT Press.

Roschelle, J., & Teasley, S. D. (1994). The construction of shared knowledge in collaborative problem solving. *NATO ASI Series F Computer and Systems Sciences, 128*, 69–69.

Roschelle, J., Digiano, C., Pea, R. D., & Kaput, J. (1999). Educational Software Components of Tomorrow (ESCOT), *Proceedings of the International Conference on Mathematics/Science Education & Technology (M/SET)*, March 1–4, 1999. San Antonio, USA.

Sandoval, W. A., & Millwood, K. (2005). The quality of students' use of evidence in written scientific explanations. *Cognition and Instruction, 23*(1), 23–55.

Schauble, L., Klopfer, L. E., & Raghavan, K. (1991). Students' transition from an engineering model to a science model of experimentation. *Journal of Research in Science Teaching, 28*, 859–882.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer, 39*(2), 25–31.

Segedy, J. R., Kinnebrew, J. S., & Biswas, G. (2012). Promoting metacognitive learning behaviors using conversational agents in a learning by teaching environment. *Educational Technology Research & Development*.

Sengupta, P. (2011). *Design Principles for a Visual Programming Language to Integrate Agent-based modeling in K-12 Science.* In: Proceedings of the Eighth International Conference of Complex Systems (ICCS 2011), pp 1636–1637.

Sengupta, P., & Farris, A. V. (2012). Learning Kinematics in Elementary Grades Using Agent-based Computational Modeling: A Visual Programming Based Approach. *Proceedings of the 11th International Conference on Interaction Design & Children*, pp 78–87.

Sengupta, P., & Wilensky, U. (2009). Learning electricity with NIELS: Thinking with electrons and thinking in levels. *International Journal of Computers for Mathematical Learning, 14*(1), 21–50.

Sengupta, P., & Wilensky, U. (2011). Lowering the learning threshold: Multi-agent-based models and learning electricity. In M. S. Khine & I. M. Saleh (Eds.), *Dynamic modeling: Cognitive tool for scientific inquiry* (pp. 141–171). New York, NY: Springer.

Sengupta, P., Farris, A. V., & Wright, M. (2012). From agents to aggregation via aesthetics: Learning mechanics with visual agent-based computational modeling. *Technology, Knowledge & Learning, 17*(1–2), 23–42.

Sherin, B. (2001). A comparison of programming languages and algebraic notation as expressive languages for physics. *International Journal of Computers for Mathematics Learning:, 6*, 1–61.

Sherin, B., diSessa, A. A., & Hammer, D. M. (1993). Dynaturtle revisited: Learning physics through collaborative design of a computer model. *Interactive Learning Environments, 3*(2), 91–118.

Smith, J. P., diSessa, A. A., & Roschelle, J. (1993). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *Journal of the Learning Sciences, 3*(2), 115–163.

Smith, D., Cypher, A., & Tesler, L. (2000). Programming by example: Novice programming comes of age. *Communications of the ACM, 43*(3), 75–81.

Soloway, E. (1993). Should we teach students to program? *Communications of the ACM, 36*(10), 21–24.

Tan, J., & Biswas, G. (2007). Simulation-based game learning environments: Building and sustaining a fish tank. In *Proceedings of the First IEEE International Workshop on Digital Game and Intelligent Toy Enhanced Learning* (pp. 73–80). Jhongli, Taiwan.

Tanimoto, S. L. (1990). VIVA: A visual language for image processing. *Journal of Visual Languages and Computing, 1*, 127–139.

Von Glaserfeld, E. (1991). Abstraction, re-presentation, and reflection: An interpretation of experience and of Piaget's approach. In L. P. Steffe (Ed.), *Epistemological foundations of mathematical experience* (pp. 45–67). New York: Springer.

White, B. Y., & Frederiksen, J. R. (1990). Causal model progressions as a foundation for intelligent learning environments. *Artificial Intelligence, 42*(1), 99–157.

Wilensky, U. (1999). *NetLogo.* Center for Connected Learning and Computer-Based Modeling (http://ccl.northwestern.edu/netlogo). Northwestern University, Evanston, IL.

Wilensky, U., & Novak, M. (2010). Understanding evolution as an emergent process: Learning with agent-based models of evolutionary dynamics. In R. S. Taylor & M. Ferrari (Eds.), *Epistemology and science education: Understanding the evolution vs. Intelligent design controversy.* New York: Routledge.

Wilensky, U., & Reisman, K. (2006). Thinking like a wolf, a sheep or a firefly: Learning biology through constructing and testing computational theories—An embodied modeling approach. *Cognition & Instruction, 24*(2), 171–209.

Wilensky, U., & Resnick, M. (1999). Thinking in levels: A dynamic systems perspective to making sense of the world. *Journal of Science Education and Technology, 8*(1).

Wing, J. M. (2006) Computational Thinking. *Communications of the ACM,* vol. 49, no.3 March 2006, pp. 33–35.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society, 366*, 3717–3725.