Using Qualitative Physics to Create Articulate Educational Software

Kenneth D. Forbus, The Institute for the Learning Sciences, Northwestern University

QUALITATIVE PHYSICS CAPTURES THE KINDS OF REPRESENTATIONS AND REASONING TECHNIQUES THAT PEOPLE USE IN DEALING WITH THE PHYSICAL WORLD. USING QUALITATIVE PHYSICS, DEVELOPERS CAN CREATE EDUCATIONAL SOFTWARE THAT IS FLUENT, SUPPORTIVE, GENERATIVE, AND CUSTOMIZABLE.

RESEARCH IN QUALITATIVE PHYSICS has many motivations, ranging from improving our understanding of human cognition to making new kinds of software systems for various applications. In particular, creating new kinds of educational software has been a motivation for qualitative physics since its inception.[1,2] This goal is now beginning to be realized, thanks to advances in qualitative physics and computer technology. Qualitative physics provides a critical enabling technology for science and engineering education and makes possible a new type of affordable, interactive educational software—*articulate software*.

The Qualitative Reasoning Group at Northwestern University's Institute for the Learning Sciences is developing architectures for articulate software. One such architecture is the *articulate virtual laboratory*, which helps students learn by engaging them in conceptual design tasks. We've used this architecture to create educational software for science and engineering, which we've deployed experimentally in several college courses.

Qualitative physics and articulate software

Qualitative physics is particularly appropriate for application to science and engineering education for two reasons:

First, qualitative physics represents the right kinds of knowledge. Much of what students learn about science in elementary, middle, and high school consists of causal theories of physical phenomena: analyzing what happens, when it happens, what affects it, and what it affects. The representational issues that are the central concern of qualitative physics are exactly those that must be addressed by domain content providers for science education software. A simulation of a rain forest, for example, must take into account evaporation, condensation, rainfall, and other physical processes.

Today's simulations leave such factors implicit in the structure of the simulation. Typically, the conceptual understanding that the simulation designer used to create the software is not available to the software's users except very indirectly, through the program's documentation. This lack of a tight coupling between concepts and their embodiment in software makes it difficult for today's educational software to explain itself. The situation gets worse as you move away from simulators to other kinds of educational software. Without the ability to formally represent the conceptual entities that a student must assimilate to understand a domain, and the ability to reason about those entities in a human-like way, educational software cannot generate explanations on demand.

Second, qualitative physics represents the right level of knowledge. The prevailing attitude is that engineering education must be heavily mathematical. This attitude seems counterproductive. The lament that students memorize formulae without understanding the broader principles being taught is familiar to instructors. Indeed, cognitive scientists

0885-9000/97/$10.00 © 1997 IEEE

have extensively documented the existence of persistent misconceptions in domains such as physics, even after a college physics course.[3]

We conjecture that students should deeply understand the qualitative principles that govern a domain—including the mechanisms, such as physical processes, and the causal relationships—before they are immersed in quantitative problems. The better textbooks do introduce ideas in qualitative terms before diving into quantitative details, but even they do not linger at the qualitative level. One reason for shortchanging the role of qualitative understanding is that instructors do not have a systematic, formal vocabulary for qualitative knowledge, making it harder to articulate than quantitative knowledge. Qualitative physics provides the representational ideas that enable the creation of such systematic, formal vocabularies. Consequently, a long-term outcome of qualitative physics research should be to help domain experts achieve a formal understanding of those aspects of their expertise that currently are described as "intuition" and "art."

Whether or not you believe that engineering education must be heavily mathematical, such claims are impossible to make about precollege science education. Students learn calculus, at best, at the end of high school and first encounter algebra only at the start of high school. Making students memorize differential equations in the guise of teaching them science simply isn't an option. Even formal algebraic models are not feasible for elementary and middle-school students. On the other hand, students are taught what parameters exist, partial information about relationships between them (for example, "this goes up when that goes down"), and when various relationships are relevant (that is, what physical phenomena the parameters are tied to). In other words, the qualitative mathematics developed in qualitative physics provides exactly the right level of language for expressing relationships between continuous properties, for precollege science students.

These two claims suggest that qualitative physics is the key to creating much smarter educational software: software whose models of the world have a better "conceptual impedance match" with people's mental models. Such software should be

- *Fluent*. It should have some understanding of the subject being taught and be able to comprehensibly communicate to students both its results and reasoning processes.
- *Supportive*. It should include a mentoring component consisting of coaches and tutors that scaffold students appropriately, taking care of routine and unenlightening subtasks, and helping students learn how to approach and solve problems.
- *Generative*. Students and instructors should be able to pose new questions and problems, rather than just select from a small prestored set of choices.
- *Customizable*. Instructors should be able to modify, update, and extend the software's libraries of phenomena, designs, and domain theories, without needing sophisticated programming skills. This would simplify maintenance and provide scalability.

WE CONJECTURE THAT STUDENTS SHOULD DEEPLY UNDERSTAND THE QUALITATIVE PRINCIPLES THAT GOVERN A DOMAIN BEFORE THEY ARE IMMERSED IN QUANTITATIVE PROBLEMS.

I call software with these properties *articulate software*.

CyclePad: an articulate virtual laboratory

Design provides a meaningful context for learning fundamental physical principles, and design activities provide powerful motivation. In designing a household refrigerator, for instance, a student will quickly discover that water makes a poor working fluid, because its saturation curve would require very low operating pressures to achieve vaporization at typical operating conditions. Design requires that students use knowledge in an integrated fashion rather than memorize isolated facts. Getting students to think in design terms leads naturally to building a strong interest in understanding complex, real-world relationships: you can ask "Why did they design it that way?" about any artifact. Design environments that provide appropriate scaffolding for students, so that they can focus on particular areas of interest, could prove invaluable for instruction in basic science as well as engineering, and could better motivate their interest in science.

Our articulate virtual-laboratory architecture addresses this need.[4] Like existing virtual laboratories (for example, Electronics Workbench and Interactive Physics), it includes a software environment for creating and analyzing designs without the expense (and sometimes danger) of creating physical artifacts. Unlike existing virtual laboratories, it provides explanation facilities and coaching, to help guide students. One example of an articulate virtual laboratory is CyclePad,[5] which we are developing for engineering thermodynamics.

How CyclePad helps. CyclePad is an intelligent learning environment that scaffolds students in the design of thermodynamic cycles (see the sidebar, "The design of thermodynamic cycles"). It addresses several problems that students have in learning about thermodynamic cycles:

- Students tend to get bogged down in the mechanics of solving equations and carrying out routine calculations. This prevents them from exploring multiple design alternatives and leads them to avoid trade-off studies (for example, seeing how efficiency varies as a function of turbine efficiency versus how it varies as a function of boiler outlet temperature). Yet without making such comparative studies, students lose many opportunities for learning. CyclePad automates routine calculations and provides tools for visualization and sensitivity analyses. (Sensitivity analysis determines how a change in one parameter affects another parameter—for example, how the boiler pressure affects the cycle's thermal efficiency.)
- Students often have trouble thinking about what modeling assumptions they need to make. For instance, unless you know the pressure drop across a heater, it is reasonable to assume that it operates isobarically. If students don't know when to make simplifying assumptions, they'll get stuck when analyzing a design.

The design of thermodynamic cycles

The analysis and design of thermodynamic cycles is the major task that drives engineering thermodynamics, aside from applications to chemistry.[1] A thermodynamic cycle is a system in which a working fluid (or fluids) undergoes a series of transformations to process energy. Every power plant, engine, refrigerator, and heat pump is a thermodynamic cycle.

Thermodynamic cycles play much the same role for engineering thermodynamics as electronic circuits do for electrical engineering: A small library of parts (in this case, compressors, turbines, pumps, heat exchangers, and so forth) are combined into networks, thus potentially generating an unlimited set of designs for any given problem. (Practically, cycles range from four components, in the simplest cases, to networks consisting of dozens of components.) One source of the complexity of cycle analysis stems from the complex nature of liquids and gases. To improve designs, subtle interactions between their properties must be harnessed. Cycle analysis answers questions such as what is a system's overall efficiency, how much heat or work is consumed or produced, and what operating parameters (for example, temperatures and pressures) do its components require. As in many engineering design problems, an important activity in designing cycles is performing sensitivity analyses. These analyses help designers understand how choices for the properties of the components and the operating points of a cycle affect the cycle's global properties.

To illustrate, consider the cycle in Figure A. Air from the atmosphere is compressed, which raises its pressure and temperature. The combustion chamber adds more heat by injecting and igniting fuel. Energy is extracted by expanding the gas through Turbine 1, and the gas is reheated and then passed through Turbine 2 to extract yet more energy. One consequence of the Second Law of Thermodynamics is that a cycle must reject some heat as waste; this cycle first rejects the heat to a second cycle, via the heat exchanger. The waste gases are then exhausted back to the atmosphere, which is represented by a cooler so that we take into account the heat lost in this transaction. The heat transferred from the gas cycle via the heat exchanger is sufficient to vaporize the working fluid (in this case water) in the lower cycle into superheated steam, which passes through Turbine 3 to extract yet more work. Finally, the steam is condensed back into water, exhausting more heat to the atmosphere, and is pumped back into the heat exchanger to complete the cycle. A thermodynamics expert would recognize this as a combined cycle, where a Brayton gas cycle with reheat drives a Rankine vapor cycle.

In thermodynamics education for engineers, cycle analysis and design generally appear toward the end of their first semester or in a second course, because understanding cycles requires a broad and deep understanding of thermodynamics fundamentals. However, even the most introductory engineering thermodynamics textbooks tend to devote several chapters to cycle analysis, and in more advanced books, the fraction devoted to cycles rises sharply. Indeed, some textbooks focus exclusively on cycle analysis.[2] Beside the intrinsic interest of thermodynamic cycles, their conceptual design provides a highly motivating context for students to learn fundamental principles more deeply than they would otherwise.

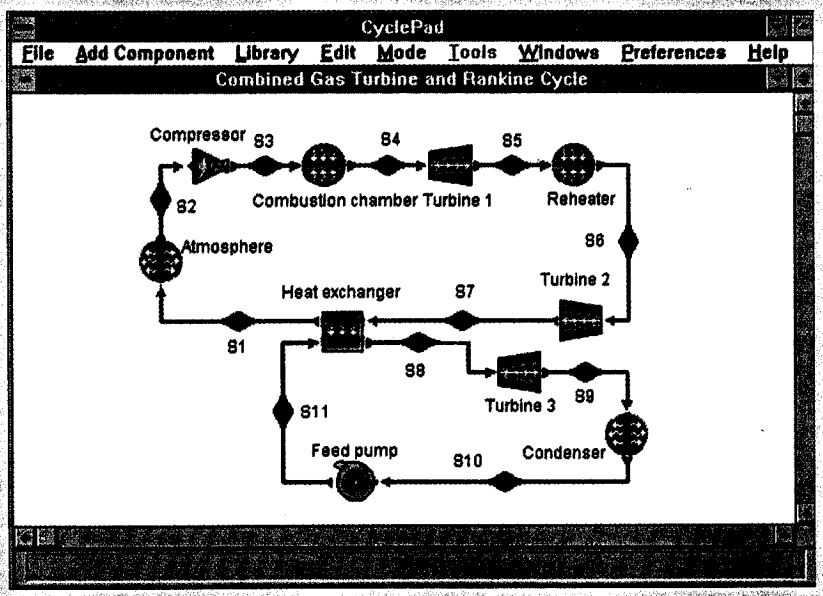

Figure A. A typical thermodynamic cycle used to generate power.

References

1. P. Whalley, *Basic Engineering Thermodynamics*, Oxford Univ. Press, Oxford, UK, 1992.
2. R.W. Haywood, *Analysis of Engineering Cycles: Power, Refrigerating and Gas Liquefaction Plant*, Pergamon Press, Oxford, UK, 1985.

CyclePad helps students keep track of modeling assumptions.

- Students often don't challenge their choices of parameters to see if their design is physically possible. For instance, a cycle that uses pumps that produce work instead of consuming it would certainly be efficient but is not, alas, possible. CyclePad detects physically impossible designs.

(These observations are based on the experience of Peter Whalley, a collaborator on CyclePad's design, who teaches engineering thermodynamics to Oxford undergraduates.)

CyclePad in action. When a student starts up CyclePad, he or she finds a palette of component types (for example, turbine, compressor, pump, heater, cooler, heat exchanger, throttle, splitter, and mixer) that can be used in the design. Components are connected together by *stuffs*, which represent the working fluid's properties at that point in the system. (Stuffs play the same role as nodes in electronic circuits.)

Once the student has put together the cycle's structure, he or she can use CyclePad to analyze the system. The student enters assumptions such as the choice of working fluid, the values of specific numerical parameters, and modeling assumptions about components. (For example, a turbine can be assumed to be either adiabatic, isothermal, or isentropic. Such modeling assumptions can introduce new constraints that might extend an analysis and new parameters—for example, the turbine's efficiency if it isn't isentropic—that must be set.)

CyclePad accepts information incrementally, deriving from each assumption as many consequences as it can. At any point, the student can ask questions by clicking on a displayed item to obtain the set of questions (or commands) that make sense for it. In addition to numerical parameters and structural

information, CyclePad displays all modeling assumptions made about a component; clicking on a component shows those modeling assumptions that are legitimate, given what is known about the system. CyclePad displays the questions and answers in English, and includes links back into the explanation system, thus providing an incrementally generated hypertext (see Figure 1).

When CyclePad uncovers a contradiction, it provide tools to resolve the problem by presenting the source of the contradiction (for example, an impossible fact becoming believed, or conflicting values for a numerical parameter) and the set of assumptions underlying that contradiction. The student can use the hypertext dialog facilities with this display to figure out which assumptions are dubious and change them accordingly.

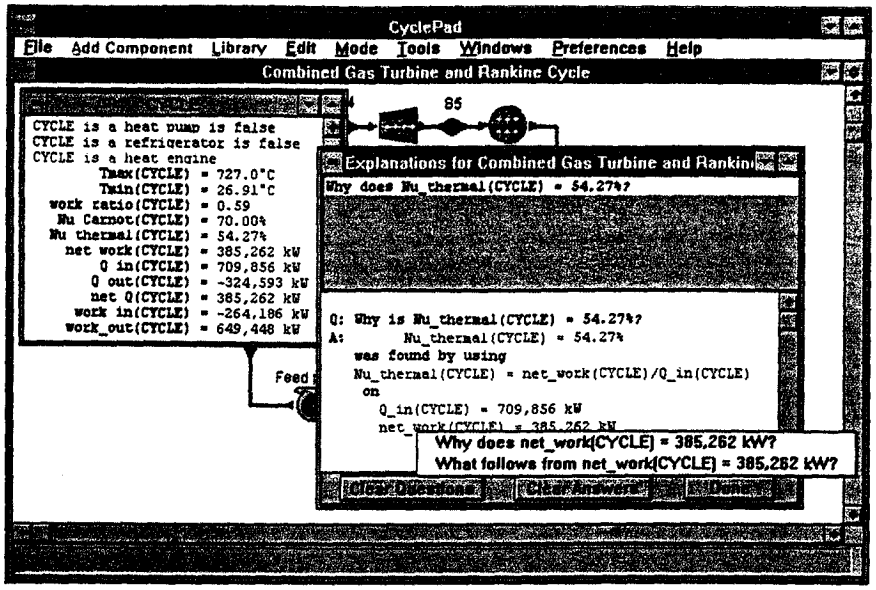

Figure 1. CyclePad provides generative hypertext explanations.

How CyclePad works. CyclePad was inspired in part by EL,[6] which analyzed analog electronic circuits. EL was one of the first systems to use constraint propagation and dependency networks to organize its reasoning, and introduced the idea of dependency-directed backtracking. CyclePad relies on several advances in the field since EL, including qualitative physics and compositional modeling. These AI techniques provide much of the scaffolding students need to carry out conceptual design tasks.

CyclePad uses compositional modeling[7] to represent the domain knowledge. Figure 2 shows a few model fragments from CyclePad's domain theory. Overall, the domain theory includes

- *Physical and conceptual entities*: components such as compressors, turbines, pumps, and heat exchangers; physical processes such as compression, combustion, and expansion; and the representations of the properties of the working fluid between the components (for example, S1 in Figure A). CyclePad's knowledge base currently contains over 29 entity definitions.
- *Structural knowledge*: the possible relationships between components, process occurrences, and the descriptions of the working fluids that connect the components. CyclePad's knowledge base currently contains 34 structural facts.
- *Qualitative knowledge*: the kinds of physical processes that can occur inside components or in the sequence of operations in an open cycle. The occurrence of physical processes places constraints on the situation's parameters; for instance, the temperature of a stuff coming into a heater cannot be higher than the temperature of the stuff leaving it. CyclePad's knowledge base currently contains definitions of five physical processes.
- *Quantitative knowledge*: equations that define relationships between the parameters of the constituents of a cycle, numerical constants (for example, molecular weights), and tables of property values for substances (for example, saturation and superheat tables). CyclePad also automatically derives equations for global properties. For example, every time the cycle's structure changes, CyclePad derives equations for net work and heat

```
(defEntity (Abstract-hx ?self ?in
                                ?out)
  (thermodynamic-stuff ?in)
  (thermodynamic-stuff ?out)
  (total-fluid-flow ?in ?out)
  (== (mass-flow ?in)
      (mass-flow ?out))
  (parameter (mass-flow ?self))
  (parameter (Q ?self))
  (parameter (spec-Q ?self))
  (heat-source (heat-source ?self))
  ((parts :cycle) has-member ?self)
 (?self part-names (in out))
 (?self IN ?in)(?in IN-OF ?self)
  ?self out ?out)(?out out-of ?self))

(defAssumptionClass
  ((abstract-Hx ?hx ?in ?out))
    (isobaric ?hx)
    (:not (isobaric ?hx)))

(defEntity (Heater ?self ?in ?out)
  (abstract-Hx ?self ?in ?out)
  (?self instance-of heater)
  (heat-flow (heat-source ?self)
             (heat-source ?self)
             ?in ?out)
  ((heat-flows-in :cycle)
     has-member (Q ?self))
  (> (Q ?self) 0.0))

(defEquation Hx-law
  ((Abstract-Hx ?hx ?in ?out))
  (:= (spec-h ?out)
      (+ (spec-h ?in) (spec-Q ?hx))))

(defEquation spec-Q-definition
  ((Abstract-Hx ?hx ?in ?out))
  (:= (spec-Q ?hx)
      (/ (Q ?hx) (mass-flow ?hx))))
```

Figure 2. A sample of CyclePad's knowledge base.

Active illustrations

The power of illustrative examples is well-known in education. Traditional media offer high authenticity but low interactivity. Textbook illustrations and posters can provide thought-provoking pictures, tables, charts, and other depictions of complex information. Movies and video can provide gripping dynamical displays. But none of these media provide interaction. Students intrigued by a picture of a steam engine in a textbook (or a movie of a steam engine) cannot vary the load or change the working fluid to see what will happen. They cannot ask for more details about explanations that they don't understand. They cannot satisfy their curiosity about how efficiency varies with operating temperatures by testing the engine over ranges of values.

The active illustrations architecture uses AI techniques to provide such interaction. An active illustration is like a hands-on museum exhibit, consisting of a virtual artifact or system and a guide who is knowledgeable about the exhibit and enthusiastically helps you satisfy your curiosity about it. Active illustrations support student explorations by letting students change parameters and relationships to see what happens. They are articulate, in that students can ask why some outcome occurred or some value holds, and receive understandable explanations that ultimately reveal fundamental physical principles and laws.

The principle component of active illustrations for dynamical systems are self-explanatory simulators.[1–3] A self-explanatory simulator combines qualitative and numerical representations to provide both accurate quantitative descriptions of behavior and conceptual explanations of it. A self-explanatory simulator can describe at every point in the simulation exactly what is happening in the simulated system and why, ranging from qualitative, causal explanations suitable for novices to sets of ordinary differential equations for an expert audience. Most important, self-explanatory simulators can be compiled automatically from domain theories in polynomial time.[4] Figure B illustrates the compilation process.

We are exploring this architecture for the domain of middle-school earth science, focusing on the processes that underlie the weather. We are building a sequence of active illustrations, starting with laboratories for exploring fundamental processes such as evaporation and phase changes, and expanding to system-level models of the hydrological cycle and the atmosphere.

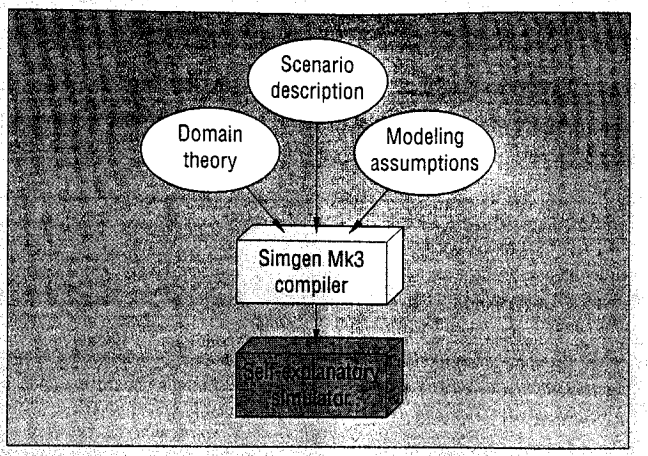

Figure B. Automatic compilation of self-explanatory simulators.

The Evaporation Laboratory: a prototype active illustration. Suppose a student is interested in how evaporation works. Because evaporation happens in everyday circumstances that are neither dangerous nor expensive to set up, it can easily be experimented with. The student sets up different jars of water, varying in width and amount of water, and measures their initial level. She places these jars on the window ledge in the classroom and looks for something else to do while waiting for the experiment's outcome. Seeing an unused computer, she starts up an active illustration on evaporation, to try to gain some insights in minutes instead of days.

The student's interaction with the simulation laboratory starts with setting up a scenario. She selects, from an on-screen catalog, a cup to use in an experiment. The cups are all the same shape and size, but they are made from a variety of materials, ranging from Styrofoam to tin to titanium and even diamond. The student chooses a Styrofoam cup because such cups are common. From another catalog, she selects an environment to place the cup in. Because it is hot outside, she selects Chicago in the summer and sets the simulator to run for four hours of virtual time. A few moments later, the simulation is finished. The student notices, by requesting a plot of how the cup's water level changes over time, a slow but measurable decline. Using the explanation system, she finds the summary of the behavior shown in Figure C1. She follows up by using the explanation system's hypertext facilities (see Figure C2).

At this point, the student conjectures that higher temperature should

```
Between 0.0 and 14400.0 seconds,
  evaporation from Cup occurs,
  flow of heat from Atmosphere to water in Cup occurs,
  there is water in liquid form in Cup
  water in Cup touches the atmosphere
(1)

In Styrofoam cup in Chicago,
mass of water in Cup can be affected by:
  water loss via evaporation from Cup
In Styrofoam cup in Chicago,
water loss via evaporation from Cup can be affected by:
  vapor pressure of Atmosphere
  saturation pressure of Atmosphere
  surface area of water in Cup
  temperature of water in Cup
(2)
```

Figure C. Using an active illustration to perform a water evaporation experiment: (1) the explanation system's summary of behavior; (2) an analysis from the explanation system's hypertext facilities.

flows into and out of the cycle. CyclePad's knowledge base currently contains 167 equations, and saturation and superheat tables for 10 substances.

- *Modeling assumptions*: simplifications that can be made about a component or process during an analysis. For instance, the pressure drop across a boiler is typically ignored in conceptual design because it is negligible for analytical purposes. Rather than stipulate a particular pressure drop, it is simpler to assume that the heater used to model a boiler is *isobaric*—that is, has no pressure drop. CyclePad's knowledge base currently contains 10 types of modeling assumptions.
- *Assumption classes*: classes that organize modeling assumptions into sets. When an assumption class is active, a model of the cycle must include one assumption from it for that model to be complete. CyclePad's knowledge base currently contains 14 assumption classes.

To interactively and incrementally derive the consequences of each student assumption, CyclePad uses antecedent constraint

lead to more evaporation. To confirm this conjecture, she runs a second simulation, using a diamond cup to increase the flow of heat from the atmosphere. Qualitatively, the behavior is the same, but diamond's higher thermal conductivity means that the diamond cup's temperature will quickly become close to the ambient temperature, and indeed leads to increased evaporation (see Figure D).

The student might continue her explorations by deciding to see what happens with the same cup on the top of a mountain, where it would be very cold, or in the tropics, where the temperature could be adjusted to be the same as in the desert but with a much higher relative humidity. These explorations can be accomplished in minutes, with reports produced for further comparison and reflection.

From laboratory to classroom. Active illustrations would be appropriate in these settings:

- *Stand-alone systems*. The student views the active illustration as a separate tool. It could be a laboratory for running experiments, such as the Evaporation Laboratory, or a training simulator.
- *Hypermedia component*. Active illustrations could be a powerful new type of media in hypermedia systems. A student might start using an active illustration included to provide a concrete example of some phenomena, and branch to the rest of the network, based on the concepts in the illustration's explanation system.
- *Virtual artifacts in shared virtual environments*. Many groups are exploring MUDs (*m*ultiuser *d*omains) and MOOs (*M*UD, *o*bject-*o*riented) as environments for students to interact with each other and instructors, in an arena designed to support learning. Because interaction is computer-mediated, such spaces provide additional opportunities for software-based coaching and assessment of student progress.

The need to explore these different settings in parallel raises interesting software-engineering issues. Our solution is to produce self-explanatory simulator runtime libraries that can be combined with code produced for particular simulators and hooked into shells that support these different settings. For producing small-footprint systems, we have created a Windows dynamic linked library that can be linked with DLLs for specific simulators. For cross-platform portability and flexibility, we also support a Common Lisp runtime library. Our Simgen Mk3 compiler (see the Simgen sidebar) produces Common Lisp simulators that can be used directly with the Lisp runtime library and produces C++ source code that can be compiled to create a simulator DLL.[4] So far, we have created two shells for active illustrations. The first is a GUI-based shell, written in C++, which provides a small-memory-footprint stand-alone system. The second is a client-server shell, with transactions mediated via the MUD Communications Protocol. The MCP allows student/simulator interactions to be broadcast through a MUD, thus making the interactions visible and available for coaching, tutoring, and evaluation software. (The MCP is a message protocol for software that needs to interoperate with MUDs. Its specification has been published on the Web.)

A target audience such as middle-school students raises a number of design issues for an active illustration. First, as I noted in the main article, middle-school students cannot be expected to understand ordinary differential equations. Consequently, we have put filters on the explanation system to hide information that would be inappropriate for this audience. (More than one teacher has suggested adding a "nerd switch" that would provide access to the full explanation system, on the grounds that early exposure to such information in the right circumstances could spark curiosity.) The kinds of queries allowed are thus very limited. For instance, the only questions students can ask about a parameter are "What can affect it?" and "What can it affect?"

We focus on the kind of causal information that students are supposed to be learning. The answers to questions about parameters, for instance, concern the existence of influences—for example, "`X can be affected by...`" in Figure C1. Although the explanation system knows the type and sign of the influence, the filter suppresses this information because it is something that the student should be learning, along with the

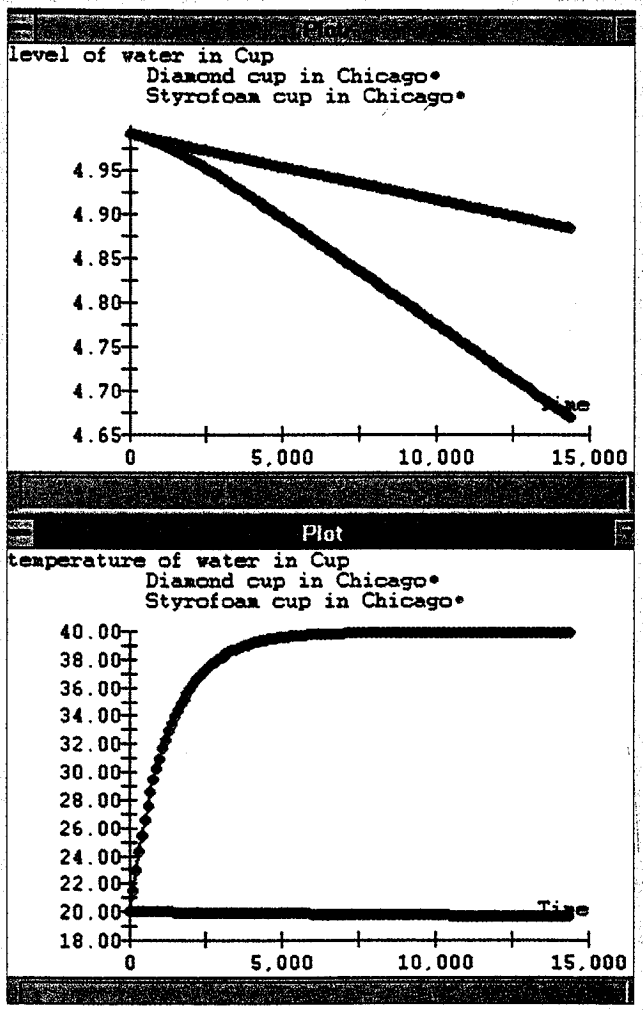

Figure D. Comparison of evaporation from Styrofoam and diamond cups, summertime in Chicago: water level (top); water temperature (bottom).

propagation, recording the derivations in a *logic-based truth-maintenance system*.[8] Upon request, CyclePad provides explanations of derived values, the indirect consequences of particular assumptions the student made, the equations that might be relevant to deriving a particular value, and other similar information. These explanation facilities exploit the dependency network created in the LTMS.

CyclePad represents explanations as *structured explanations*, an abstraction layer between the reasoning system and the interface. The reason for this layer is that the reasoning system must be optimized for performance, while the interface must be optimized for clarity, and these goals are often in conflict. The structured-explanation layer provides summarization, hiding aspects of how the reasoning system works that are irrelevant to the student. It also provides reification, making dependencies explicit that would otherwise be implicit, such as the various methods that could be used to derive a desired parameter.

Automating the tedious calculations involved in thermodynamic equations and

relative magnitudes of various effects. (By type of influence, we mean whether the causal relationship partially specifies a functional dependency or a rate dependency. Qualitative Process theory provides form representations for such casual relationships.[5] Partial information about functional dependencies is expressed via *qualitative proportionalities*, and partial information about rate dependencies is expressed via *direct influences*.)

A second issue in creating simulators for students is the problem of initialization. Providing a large menu of numerical and logical parameters, even in the cleanest, well-organized GUI, can easily lead to bewilderment. We simplify this process by using a metaphor from drama—the idea of a *prop*. A prop on a stage represents something in the imagined world. In our simulators, props represent a coherent subset of the simulator's parameters that naturally make sense to consider together.

Each simulator has a set of *catalogs*, each catalog containing props that impose different constraints on a particular subset of the simulator's parameters. The Evaporation Laboratory, for instance, has two catalogs: cups and environments (see Figure E). The choice of cup constrains its shape and dimensions, as well as its thermal conductivity (for example, the thermal conductivity of diamond is orders of magnitude higher than most common materials). The choice of environment constrains the atmosphere's temperature, pressure, and vapor pressure, as well as the limits over which these parameters can vary. (Although Las Vegas could, in theory, get colder than the top of Mt. Everest, it would be very surprising. Providing constraints that prevent two props from being identical in the simulator helps maintain the suspension of disbelief.)

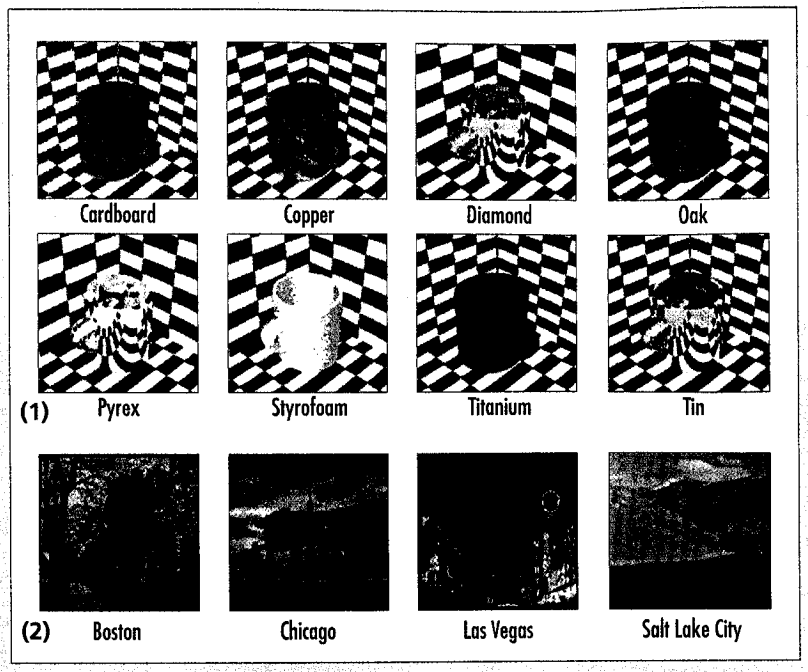

Figure E. Two catalogs used in the Evaporation Laboratory: (1) cups (students can change the amount and temperature of the water for whichever cup they choose); (2) environments.

In addition to solving the technical problem of setting up a simulation, props should also provide pedagogical benefits by helping the student see relationships between physical objects and circumstances and their properties. It also provides a simple path to customization. For example, adding props representing familiar objects and situations (such as a student's favorite cup or hometown) can make software more engaging.

Deployment and other plans. We demonstrated the Evaporation Laboratory at a CAETI meeting in March 1996. We used a client-server version of the software, with communications routed to a MUD so that evaluation agents and other software could listen in. The stand-alone version of the Evaporation Laboratory was delivered to the CAETI testing facilities in April, as part of the process of migrating the software to Dodea test schools. We are making the stand-alone version available for experimental use in schools.

We are creating several new simulations, both for earth science and for other domains. As with CyclePad, we are gathering feedback from teachers that is driving the evolution of the interfaces for the shells. We are collaborating with other members of the CAETI community to build simulation coaches that can help students set up and interpret experiments in a MUD-based environment. (A client/server MUD version with coaching, using lightweight tutor agents[6] to provide advice and evaluative feedback on the design of simulation experiments, was demonstrated in November 1996. We provided the client-server simulation system, Mark Shirley and Daniel Bobrow of Xerox PARC developed the MUD communications system, and Ken Koedinger of CMU developed the tutor agent.)

The MUD environment provides some interesting challenges. MUDs are almost entirely text-based. Limitations in communications bandwidth over most of the planet, and limited education budgets, mean that many MUD-based learning spaces will continue to be text-based for years to come. Consequently, a text-based user interface for active illustrations, in the form of a MUD-based software robot, can open up this technology to an even broader audience. We also suspect that text-based bots could provide some advantages that complement those available with GUIs. First, text that engages a student's imagination provides better imagery than any graphics technology can. Second, the metaphors for interaction with bots are different: the bot can act as an assistant, carrying out experiments in a virtual environment, or as a purveyor of phenomena, helping to spark a student's interest. We are creating such a softbot, using Sibun's Salix natural-language-generation techniques to provide explanations.[7] A key issue in creating a useful softbot is handling discourse well enough in a MUD-based environment to keep students engaged.

References

1. F. Amador, A. Finkelstein, and D. Weld, "Real-Time Self-Explanatory Simulation," *Proc. 11th Nat'l Conf. Artificial Intelligence*, AAAI Press/MIT Press, Cambridge, Mass., 1993, pp. 562–567.
2. K. Forbus and B. Falkenhainer, "Self-Explanatory Simulations: An Integration of Qualitative and Quantitative Knowledge," *Proc. Eighth Nat'l Conf. Artificial Intelligence*, Vol. 1, AAAI Press/MIT Press, 1990, pp. 380–387.
3. Y. Iwasaki and C. Low, "Model Generation and Simulation of Device Behavior with Continuous and Discrete Changes," *Intelligent Systems Eng.*, Vol. 1, No. 2, 1993.
4. K. Forbus and B. Falkenhainer, "Scaling up Self-Explanatory Simulators: Polynomial-Time Compilation," *Proc. IJCAI-95*, Morgan Kaufmann, San Francisco, 1995.
5. K. Forbus, "Qualitative Process Theory," *Artificial Intelligence*, Vol. 24, 1984, pp. 85–168.
6. S. Ritter and K.R. Koedinger, "Towards Lightweight Tutoring Agents," *Proc. Seventh World Conf. Artificial Intelligence in Education*, Assoc. for the Advancement of Computing in Education, Charlottesville, Va., 1995, pp. 91–98.
7. P. Sibun, "Generating Text without Trees," *Computational Intelligence*, Vol. 8, No. 1, 1992, pp. 102–122.

explaining clearly how the student's assumptions were used provides substantial scaffolding. Students can focus on the thermodynamic consequences of their assumptions, rather than on using their calculators to solve routine equations. The LTMS also provides a useful mechanism for detecting and recovering from contradictory assumptions. For instance, if the parameters supplied by the student imply a violation of physical laws (for example, that a turbine consumes work rather than generates it), the LTMS alerts the student, also providing the subset of responsible assumptions for correction.

CyclePad provides other analysis tools besides constraint propagation. It automates sensitivity analyses. Instructors view such analyses as important for gaining a deeper appreciation of the domain. To automate these analyses, CyclePad uses the dependency network in the LTMS to identify relevant parameters and to automatically derive the necessary equations. CyclePad provides visualization tools that reveal how part of the cycle contributes to its overall performance. Graphical information about the bounds of available property tables and, in some cases, automatically generated T-S (temperature versus entropy) diagrams are also available. In addition, CyclePad includes an online help system that describes the program's operation and knowledge.

Lessons learned. CyclePad incorporates substantial domain knowledge and applies it to a complex task under tight computational constraints, so it provides interesting evidence concerning the utility of various AI techniques for educational software.

Compositional modeling can work well with large, integrated qualitative/quantitative domain theories. Although automatic model formulation is unnecessary in this task, the representations of assumption classes and logical constraints between modeling assumptions provide valuable leverage in organizing an analysis.

Qualitative representations can provide useful reality checks even in highly quantitative tasks. Besides detecting physically inconsistent designs, qualitative descriptions of physical processes provide grounds for explanations.

Numerical constraint propagation is very effective. An alternate approach would have been to use one of the many commercial symbolic mathematics packages, such as MathCAD or Mathematica. Although such packages have their uses, they are poor choices as components in articulate software. They do not provide explanation facilities, they demand substantial computational resources, and their cost would make experimentation and deployment in schools difficult. "Rolling your own" seems still to be the best choice for educational software.

Logic-based truth maintenance systems provide the right mix of logical power and efficiency. The ability to use arbitrary clauses instead of being restricted to Horn clauses (as in a justification-based truth maintenance system) facilitates the expression of relationships between modeling assumptions. The linear-time reasoning process of an LTMS provides freedom from the exponential blowups that can happen with an assumption-based truth maintenance system. However, the monotonic growth of memory consumption in standard TMS algorithms proved unacceptable for CyclePad, because retracting an assumption can result in the retraction of hundreds of consequences. Consequently, we invented a garbage collector for facts and clauses for the LTMS.[9]

QUALITATIVE PHYSICS CAN BE THE CORE TECHNOLOGY FOR CREATING NEW KINDS OF SMARTER EDUCATIONAL SOFTWARE, BECAUSE IT CAPTURES THE KINDS OF REPRESENTATIONS AND REASONING TECHNIQUES THAT PEOPLE USE IN DEALING WITH THE PHYSICAL WORLD.

Generative hypertext provides a simple but powerful explanation system. Ehud Reiter and Chris Mellish have suggested that sophisticated natural-language-generation techniques are unnecessary in many applications.[10] So far, this has been the case with CyclePad. The ability to automatically generate hypertext in response to a user's questions obviates the need for discourse planning. Also, the task's fixed nature means that we can postpone issues such as selecting the appropriate level of detail in an explanation. Hypertext lets users select how much they want to know about a topic. Because CyclePad generates the hypertext only on demand, it avoids many navigation problems common in fixed hypertexts.

Deployment and plans for CyclePad. To date, we have used CyclePad on an experimental basis with students from Northwestern University, Oxford University, and the US Naval Academy. These formative evaluations have led to substantial changes in the interface and a variety of new features that make the software more useful. CyclePad is now stable enough that it has been used in required classroom assignments. (CyclePad is written in Allegro Common Lisp for Windows, exploiting the ability to produce royalty-free runtime systems to produce a distribution that fits on two floppies. For an HTML version of CyclePad's help system, access *http://www.qrg.ils.nwu.edu.*) The most extensive use has been at the Naval Academy, where, for example, 17 students have used it for class assignments and term projects in an energy-conversion course taught by C. Wu in the spring semester of 1996. We plan to use it extensively with both introductory and advanced students at all three universities in the 1996–1997 academic year in further studies. We also plan to refine its design based on these experiences and the suggestions of our thermodynamics collaborators.

We are also adding coaching to the software for further scaffolding. Students need two kinds of advice in these design tasks. First, they need help in analysis. We are using a combination of rules and special-case procedures for analysis coaching. For instance, one common source of trouble for students is selecting values outside the range of the property tables. Such circumstances trigger rules that display the offending assumptions and a chart of the relevant property table. The chart shows what information is available and where the student's choices are with respect to what is known, so that the student can revise his or her assumptions to fit the available data.

Second, students need help improving their design to meet their project's goals. We plan to use a case-based coach for design.

With our thermodynamics collaborators, we will create the library of cases. We'll implement the coach using analogical reasoning tools that our group developed originally as cognitive simulations. The retrieval component will be MAC/FAC,[11] a model of similarity-based reminding. MAC/FAC's advantage should be that we will not have to hand-index the case library's elements—the representations automatically computed by CyclePad will provide the necessary information. SME,[12,13] our simulation of analogical mapping, will adapt a case to the student's current problem. The candidate inferences that SME produces as part of a mapping should provide suggestions for how the information in the case can help improve the student's design.

We are also designing a second laboratory, aimed at teaching intuitive notions of feedback control to high-school and college students. This domain complements engineering thermodynamics well: In engineering thermodynamics, the analyses are steady state, and the complexity arises from the complex nature of substances and the processes they undergo. In feedback control, a controller's components can be viewed functionally (abstracting away from whatever technology is used to implement them), and the complexity arises from the dynamic interaction of simple parts. Although the analysis tools and methods will be very different for this domain, we believe that the same articulate virtual laboratory architecture will work for it, too.

FOR COUNTRIES TO REMAIN COMpetitive in the world economy and have a technologically literate population that can govern itself wisely, high-quality education—particularly in science and engineering—is essential. One way to improve education is by improving educational software. Qualitative physics can be the core technology for creating new kinds of smarter educational software, because it captures the kinds of representations and reasoning techniques that people use in dealing with the physical world. Using qualitative physics, we should be able to create truly articulate educational software. Although much research remains, our experience with articulate virtual laboratories and with active illustrations (see the "Active illustrations" sidebar) indicates that this approach is very promising.

What will it take for qualitative physics to become widespread as a component in educational software? Three factors are essential:

- *Off-the-shelf domain theories.* A good domain theory provides material that can

Simgen

The original motivation for self-explanatory simulators was the Steamer project, one part of which attempted to retrofit qualitative, causal explanation systems to a hand-built Fortran simulation program. Such training simulators are hard to build and, once built, are typically opaque, making such retrofits difficult at best. But suppose the simulator is written by a compiler that understands the domain in the same way that an expert simulation author for that domain understands it. Then, simulators become cheaper to build, and retrofitting is unnecessary because the compiler's understanding can be embedded into the simulator itself.

The explanatory capabilities of self-explanatory simulators arise from two components. The first is a structured explanation system: a conceptual, declarative model of the situation being simulated, expressed in qualitative and quantitative terms. It is derived from, but is not identical to, the compiler's qualitative understanding of the situation. For instance, antecedents are simplified both to insulate the user from low-level details of how the domain theory was formulated and to focus on critical information. The second is a concise history, a recording of how important qualitative properties vary during a simulation. The concise history, combined with the structured explanation system, is sufficient to reconstruct the complete causal account and mathematical model at every instant during a simulation. Because the concise history is interval-based, the storage cost of maintaining it depends on the behavior's qualitative complexity, rather than whatever time step the simulation uses.

Evolution

The original self-explanatory simulator compiler (Simgen Mark 1) used *envisioning*, a form of qualitative simulation, to do the necessary qualitative reasoning. Because envisioning explicitly generates every globally consistent state, this compiler produced simulators with very stringent self-monitoring. If, for example, the simulation's numerical parameters veered from the set of legal qualitative states or suggested a qualitatively impossible transition, the simulators would detect this and complain. Unfortunately, the price for this useful capability was too high: Envisioning is generally exponential, which led to long compilation times, so this compiler could only produce simulators for systems containing approximately 10 state parameters. The second-generation compiler (Simgen Mark 2) did not use qualitative simulation at all. However, it still carried out transitivity calculations to create additional self-monitoring, using an assumption-based truth maintenance system. Although this compiler could produce simulators for systems containing a few hundred parameters, the exponential behavior of the ATMS for larger systems made it impractical for large simulations. The current-generation compiler (Simgen Mark 3) uses a logic-based truth maintenance system and symbolic evaluation instead of an ATMS, yielding polynomial-time performance, at the cost of some compile-time error checking. Simgen Mark 3 can generate simulators with thousands of parameters, which is the scale of many interesting training simulators (for example, shipboard propulsion systems and power plants).

This evolution illustrates two important points. First, the tendency has been to identify qualitative reasoning with one specific type of qualitative reasoning—namely, qualitative simulation. This misconception makes it easy to miss what might be some of the most interesting applications of qualitative reasoning. In the task of simulator generation, qualitative reasoning is still crucial. Qualitative reasoning identifies what phenomena are relevant, suggests what qualitative and mathematical models are appropriate, and provides the causal orderings, relevance conditions, and consistency tests that structure the simulator's numerical component. But qualitative simulation is unnecessary. Identifying the appropriate forms of qualitative reasoning for a task can provide orders-of-magnitude improvement when scaling up. The second point is that each step of Simgen's evolution was driven by comparisons of the compiler's operation and the code it produced with what human programmers seem to do and what their code looks like. Informal cognitive analyses of tasks can lead to substantial benefits, even if the goal of an AI project is purely engineering.

be used to build an entire class of models, automatically or semiautomatically. The ability to automatically build models and to deeply reason about them is what makes articulate software generative. However, creating domain theories is still difficult and something of an art. It requires deep domain expertise, the ability to be explicit about this expertise, and experience in representing this knowledge formally. Libraries of domain theories that require minimal customization are a critical fuel for creating articulate software.

- *Off-the-shelf reasoning and representation modules.* Although several publicly available qualitative-reasoning systems exist, none is particularly suitable for embedding in educational software. As we learn more about how to create articulate software, we should be able to identify reusable modules that can help generate new systems more rapidly. One candidate module type is a structured-explanation system, such as we used in both CyclePad and the Evaporation Laboratory (see the "Active illustrations" sidebar). In addition, tools that help domain experts build and test representations would expand the community that could contribute domain theories.
- *Authoring environments for specific architectures.* Assuming these architectures prove their worth educationally, the process of creating new software within each architecture must be streamlined so that researchers are no longer in the loop. One way is to create authoring environments that enable curriculum developers and experienced teachers to create new software, using off-the-shelf domain theories.

The ability to automatically formulate models, generate explanations, and generate simulations provides new capabilities for educational software, while improving the economics of producing it. If we collaborate with teachers, curriculum designers, and educators, we could develop software that helps improve science and engineering education substantially.

Acknowledgments

The Computer Science Division of the Office of Naval Research supported the basic research on self-explanatory simulators and compositional modeling. The ONR's Cognitive Science Division supported the basic research on analogical matching and retrieval. CyclePad's initial development was supported partly by a grant from the Science and Engineering Research Council in the UK. The Advanced Applications of Technology program of the EHR directorate of the National Science Foundation supported research on articulate virtual laboratories. The Defense Advanced Research Projects Agency of the US Department of Defense, under the Computer Education and Training Initiative, supported research on educational applications of self-explanatory simulations.

CyclePad's knowledge base has been developed in collaboration with Peter Whalley, of Oxford University. Other subject-matter collaborators and instructors involved with CyclePad include David Mintzer, Siavash Sohrab, and Michael Brokowski (Northwestern University); and C. Wu and Sheila Palmer (USNA). John Everett and Andy Bachmann have provided substantial programming support for CyclePad. John DeMastri, Aaron Thomason, and Mike Oltmans created the C++ libraries for self-explanatory simulators and the C++ runtime system. Penelope Sibun is responsible for natural-language explanation generation and softbot design.

References

1. J.S. Brown, R. Burton, and J. de Kleer, "Pedagogical, Natural Language, and Knowledge Engineering Techniques in SOPHIE I, II, and III," in *Intelligent Tutoring Systems*, D. Sleeman and J.S. Brown, eds., Academic Press, San Diego, 1982, pp. 227–282.
2. J. Hollan, E. Hutchins, and L. Weitzman, "STEAMER: An Interactive Inspectable Simulation-Based Training System," *AI Magazine*, Vol. 5, No. 2, 1984, pp. 15–27.
3. D. Gentner and A. Stevens, eds., *Mental Models*, Lawrence Erlbaum Associates, Mahwah, N.J., 1983.
4. K. Forbus, "Three Articulate Software Architectures for Science and Engineering Education," to be published as an ILS tech. report, Inst. for Learning Sciences, Northwestern Univ., Evanston, Ill.
5. K. Forbus and P. Whalley, "Using Qualitative Physics to Build Articulate Software for Thermodynamics Education," *Proc. 12th Nat'l Conf. Artificial Intelligence*, Vol. 2, AAAI Press/MIT Press, Cambridge, Mass., 1994, pp. 1175-1182.
6. R.M. Stallman and G.J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," *Artificial Intelligence*, Vol. 9, 1977, pp. 135–196.
7. B. Falkenhainer and K. Forbus, "Compositional Modeling: Finding the Right Model for the Job," *Artificial Intelligence*, Vol. 51, Nos. 1–3, Oct. 1991, pp 95–143.
8. K. Forbus and J. de Kleer, *Building Problem Solvers*, MIT Press, 1993.
9. J. Everett and K. Forbus, "Scaling up Logic-Based Truth Maintenance Systems via Fact Garbage Collection," *Proc. 13th Nat'l Conf. Artificial Intelligence*, Vol. 1, AAAI Press/MIT Press, 1996, pp. 614–620.
10. E. Reiter and C. Mellish, "Optimizing the Costs and Benefits of Natural Language Generation," *Proc. IJCAI-93*, Morgan Kaufmann, San Francisco, 1993, pp. 1164–1169.
11. K. Forbus, D. Gentner, and K. Law, "MAC/FAC: A Model of Similarity-Based Retrieval," *Cognitive Science*, Vol. 19, No. 2, Apr.–June, 1995, pp. 141–205.
12. B. Falkenhainer, K. Forbus, and D. Gentner, "The Structure-Mapping Engine: Algorithm and Examples," *Artificial Intelligence*, Vol. 41, 1989, pp. 1–63.
13. K. Forbus, R. Ferguson, and D. Gentner, "Incremental Structure-Mapping," *Proc. Cognitive Science Soc.*, Lawrence Erlbaum Associates, 1994, pp. 313–318.

Kenneth D. Forbus is a professor of computer science and education at Northwestern University. His research interests include qualitative physics, analogy and similarity, cognitive simulation, reasoning system design, and educational software. He received an SB and an SM in computer science and a PhD in artificial intelligence, all from MIT. He is a Fellow of the AAAI and serves on the editorial boards of *Artifical Intelligence*, the *Journal of AI Research*, and the AAAI Press. Contact him at the Inst. for the Learning Sciences, 1890 Maple Ave., Evanston, IL 60201; forbus@ils.nwu.edu; http://www.qrg.ils.nwu.edu.