The Role of Models in Computer Science

Abstract

Taking Brian Cantwell Smith's study, "Limits of Correctness in Computers," as its point of departure, this article explores the role of models in computer science. Smith identifies two kinds of models that play an important role, where *specifications* are models of problems and *programs* are models of possible solutions. Both presuppose the existence of *conceptualizations* as ways of conceiving the world "in certain delimited ways." But *high-level* programming languages also function as models of virtual (or abstract) machines, while *low-level* programming languages function as models of causal (or physical) machines. The resulting account suggests that sets of models embedded within models are indispensable for computer programming.

Keywords: specifications, programs, programming languages, function, model, representation, abstraction, conceptualization.

The role of models in computer science appears to be even more pervasive than has been generally acknowledged, even by sophisticated students of the discipline. In his important study, "Limits of Correctness in Computers" (1985/95), for example, Brian Cantwell Smith remarks that the relationship between a *program* and its *specification* is that of one model (the program) to another (its specifications), where even the existence of formal proofs that programs satisfy their specifications cannot guarantee that a system controlled by that program will do what it is supposed to do should that program be executed, because the specifications may or may not reflect an appropriate relationship to the world.

Smith characterizes "specifications" as formal descriptions (usually in a formal language) of what qualifies as proper behavior for a certain system,

"The Role of Models in Computer Science" by James H. Fetzer,
The Monist, vol. 82, no. 1, pp. 20–36.

while "programs" are formal descriptions (in a programming language) of precisely how that behavior is to be achieved (Smith 1985/95, p. 463). Thus, as it may be expressed, the specifications specify the outputs O that are supposed to be derivable from the inputs I when the system is operating properly, where the program specifies the sequence of instructions P by means of which those outputs are derived from those inputs. The program might therefore be characterized as a *function* that maps specific values within the range of a variable called "input" onto specific values in the domain of a variable called "output."[1]

Smith emphasizes that computers, like people, are systems that participate in the real world by taking actions that have consequences, where one of their most important features is "that we plug them in." Thus, for example, he says,

> They are not, as some theoreticians seem to suppose, pure mathematical abstractions, living in a pure detached heaven. They land real planes at real airports; administer real drugs; and—as you know all too well—control real radars, missiles, and command systems. Like us, in other words, although they base their actions on models, they have consequences in a world that inevitably transcends the partiality of those enabling models. Like us, in other words, and unlike the objects of mathematics, they are challenged by the inexorable conflict between the partial but tractible model, and the actual but infinite world (Smith 1985/95, p. 461).

In this passage and elsewhere, Smith drives home the point that computers are complex systems that inhabit the real world and interact with other things in the real world, with consequences that are capable of affecting life and death.

It follows that computers are not merely abstract entities, but real causal systems.[2] If programs are to be envisioned as functions mapping values from a range onto a domain, therefore, then those functions must be entertained as causal in character.[3] Since most programmers utilize "higher-level" programming languages, such as Pascal, LISP, and Prolog, which are related to physical computers by means of interpreters and compilers, however, these languages appear to qualify as models of yet another kind. And reliance upon programming languages at the machine or assembly language level seems to imply the existence of models of yet another distinctive kind. The role of models in computer science extends far beyond that of specifications and programs.

The purpose of this article, therefore, is to pursue the leads provided by Smith's valuable study in an attempt to further elucidate the nature and role of models in computer science. It will become evident that at least four kinds of models pervade computer science, including those afforded by programming languages themselves. Further consideration will be given to whether Smith is right in his belief that, "at this point in intellectual history," we have no theory of the relationship of models to the world. The answer to this question depends upon drawing a distinction—familiar within the philosophy of science—between the context of discovery and the context of justification. But I should begin with some general reflections regarding the nature of models themselves.

1. What are Models?

The concept of a model (what a model is, the kind of thing a model can be) appears to be somewhat vague and more than a little ambiguous. Models can range from *physical replicas* that resemble what they model (such as a plastic model of a battleship) and *working models* that instantiate the abilities of the things they model (such as a battleship prototype) up to *abstract models* that preserve selected aspects of the thing modeled while disregarding others (such as blueprints for the construction of a new battleship). These distinctions are not necessarily exclusive, however, and sometimes may even appear to be more a matter of difference of degree rather than one of a difference in kind.

Like many others, I can vividly recall, as a young man, obtaining a kit for constructing a plastic model of the battleship *Missouri*. The kit contained at least two crucial ingredients and, if you were lucky, also a third. These were, first, the parts of the model; second, the instructions for putting the parts together; and third, a tube of glue. If there were too few or too many parts, of course, or if the instructions were missing, you might have wanted to take it back and exchange it for another kit. The crucial consideration turned out to be that there should be parts of the model corresponding to parts of the thing modeled, where these could be put together to create a similar arrangement.

The underlying conception was that, at an appropriate level of detail and scale, there should be a one-to-one correspondence between the parts of the model and the parts of the thing modeled, on the one hand, and that, as long as the model was assembled in accordance with the instructions, the relations between the parts of the model should correspond with those

of their counterparts on the thing modeled, on the other. The strong relationship that is involved here, of course—a one-to-one correspondence that is relation-preserving—is known as an *isomorphism*, which is a concept familiar to mathematicians in relation to abstract contexts, but one that applies to physical contexts as well.

Appropriately conceived, isomorphic relations may be found in many kinds of models, including the blueprints for a new construction and even the prototype of a new weapon. Considerations of scale and of level of detail turn out to be fundamental to models of all three kinds. In the case of physical replicas, for example, there would be many functional differences, insofar as a model of the *Missouri* could fit into a bathtub but would lack the spots of rust and chips of paint of the real thing. A working model, by contrast, would need to possess the functional capabilities of the real thing, where the limiting case would be a physical replica with the same functional capabilities, i.e., another battleship.

2. Smith's Conception

Smith acknowledges models of all of these kinds, while emphasizing the role of models within computer science specifically in the form of "representations":

> To build a model is to conceive of the world in a certain delimited way. To some extent you must build models before building any artifact at all, including televisions and toasters, but computers have a special dependence on these models: *you write an explicit description of the model down inside the computer*, in the form of a set of rules or what are called *representations*—essentially linguistic formulae encoding, in the terms of the model, the facts and data thought to be relevant to the system's behavior. It is with respect to these representations that computer systems work (Smith 1985/95, p. 460).

Thus, to drive his point home, Smith suggests that it is this feature that distinguishes computers from other kinds of machines: "they run by manipulating representations, and representations are always formulated in terms of models. . . . [Thus] there is no computation without representation" (Smith 1985/95, p. 460).

To the extent to which *programs per se* are envisioned as "representations," there would appear to be scant room to take exception to Smith's position here. As functions that map specific values of "input" variables onto specific values of "output" variables, they do indeed provide the *sets*

of rules that determine the behavior of computer systems. To the extent to which they are based upon the use of *models*, moreover, it seems apparent that, in the case of computers, there is indeed a "special dependence" upon these models, insofar as digital machines—as opposed to abaci, by comparison—could not perform without them: absent programs, these machines would not be able to process "inputs" into "outputs."

On the other hand, it may be appropriate to ask for whom sets of rules that are loaded into machines (the form in which they exert their causal influence) function as "representations." The case of the abacus becomes interesting here, since the sets of rules by means of which their operations are performed are in the heads of their users. It remains correct, no doubt, that their users manipulate these rows of beads in accordance with models that infuse them with meaning, but it should be obvious that, in this case, at least, whatever meaning these beads possess as representations are to be found in the heads of those who use them. These representations presuppose the existence of interpreters or minds.

This suggests that it may be crucial to distinguish between representations that are meaningful *for use by a machine* and representations that are meaningful *for the users of those machines* (Fetzer 1988, p. 137). Representations do not have to be meaningful for these machines to perform their designated functions, which reflects the conception of computing machines as mark-manipulating (or "string processing") systems. So long as those machines are capable of manipulating those marks (strings) in accordance with their programs, the marks (strings) they manipulate need not be meaningful for those machines themselves. It may be sufficient that the marks (strings) they process are meaningful to their users.

3. Representations

Indeed, Smith's discussion of the role of models in computer science can be easily amended to take this distinction into account, since his position does not crucially depend on whether machines are the possessors of mentality. It does suggest, however, that *representations* depend upon the existence of interpretations, interpreters, or minds, which may be properties of their programmers and users, as the abacus exemplifies. The slogan, "no computation without representation," therefore, might be further extended to encompass the slogan, "no representation without interpretation," which implies there is "no computation without interpretation" by means of interpreters or minds (Fetzer 1994, 1998).

Insofar as Smith considers specifications and programs as "models" and as "representations," it should be evident that his concepts are somewhat broader than those associated with isomorphic representations. In order to illustrate what he has in mind, Smith advances a diagram (Figure 1), which includes (on the left) a box labeled "COMPUTER" and (on the right) a sketch of a house as an occupant of the "REAL WORLD." Mediating between them is an oblong box labeled "MODEL." Smith thus describes the connections between the three as follows:

> You are to imagine a description, program, computer system (or even a thought—they are all similar in this regard) in the left hand box, and the very real world in [sic] the right. Mediating between the two is the inevitable model, serving as an idealized or preconceived simulacrum of the world, in terms of which the description or program or whatever can be understood. One way to understand the model is as the glasses through which the program or computer looks at the world: it is the world, that is, as the system sees it (though not, of course, as it necessarily is) (Smith 1985/95, p. 462).

The sketch of a house as an occupant of the REAL WORLD, of course, is not "in" a box but rather "on" the right, where the openness of the right-hand side is meant to convey that both the MODEL and the COMPUTER are also in the REAL WORLD.

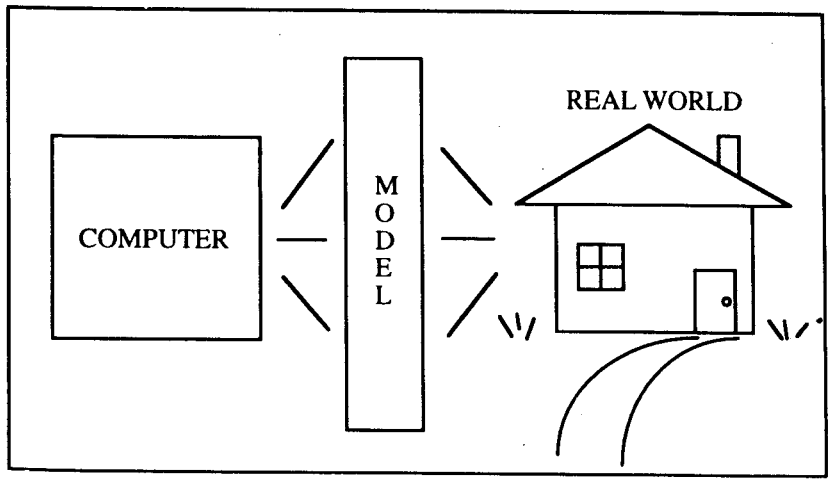

Figure 1. *Smith's Diagram* (Smith 1985/95).

The suggestion that the model is "the glasses through which the program or computer looks at the world" should be interpreted metaphorically rather than literally, since it seems to be investing inanimate objects with human (or animal) properties in the spirit of *anthropomorphism*. Nothing appears to be lost in relation to the content of Smith's contentions by refashioning his claims as meaning that the performance of the computer is causally affected by the program that it executes, where that program in turn has been designed as a model of the world *vis-à-vis* the presumptive interaction of the computer-executing-the-program-in-the-world. The only literal *glasses* required may be those worn by programmers.

That descriptions, programs, computer systems, or even thoughts may occupy the left-hand box suggests that they, too, are "models" in some appropriate sense, a sense that does not appear to be confined to the somewhat narrow constraints of isomorphic relations. Descriptions, such as "Snow is white" or 'The computer is on the table," for example, do not—at least, in any obvious way—satisfy what would be required of physical replicas, working models, or even abstract models as they have been discussed above. Thoughts, at least when they are conveyed by means of language, are obviously comparable. It thus appears plausible that, in Smith's language, the word 'model' means the same thing as 'representation'.

4. Models as Signs

This makes sense from the perspective of the theory of signs advanced by Charles S. Peirce, who envisioned *a sign* as a something that stands for something (else) in some respect or other for somebody (see Fetzer 1990, 1991/96). It should be observed, however, that things function as signs (in Peirce's sense) only relative to sign-users, who have the ability to use signs of those kinds and are not incapacitated from the exercise of that ability, which is the property that makes them *conscious* with respect to signs of those kinds. Then *cognition* occurs as the effect of causal interaction between a sign of a certain kind and sign-user who is conscious with respect to signs of that kind in relation to the user's context.

Several aspects of a Peircean account of minds as sign-using (or "semiotic") systems are relevant here. Peirce, for example, distinguished between three kinds of signs, namely: *icons* as signs that resemble what they stand for; *indices* as signs that are causes-or-effects of what they stand for; and *symbols*, as signs that are merely habitually associated with that for which they stand. Of the three kinds of models discussed above, physical replicas appear to be iconic models that resemble what they stand for, while working models are indexical as well as iconic in resembling their causal capacities. Abstract models, such as blueprints, attain resemblance relations by means of representations that are partially symbolic.

Indeed, in harmony with the broad conception Smith appears to have in mind, it should be clear that linguistic descriptions are representations that commonly stand for that for which they stand on the basis of habitual associations between signs and things. The most familiar examples of these linguistic symbols are the words that occur in ordinary languages, including English, French, and German. Words such as 'chair' and 'horse', for example, neither resemble nor are causes-or-effects of that for which they stand, but are habitual associations that usually have their origins in (spoken or written) linguistic customs, traditions, and practices, which, when reinforced by institutions, assume the status of conventions.

It should therefore be apparent that "sign" and "representation" are similar in meaning and that the use of "model" as a synonym is not inappropriate. Without elaborating the matter in detail, let us also assume that the meaning of signs ultimately consists of *concepts*, understood as habits of thought and as habits of action within specific contexts (Fetzer 1990, Fetzer 1991/96). Then when Smith characterizes specifications and programs as "models," we shall understand thereby that specifications and programs stand for certain things in certain respects for those who use and understand them, precisely because of their standing as signs. For something to function as a sign, there must be a counterpart set of sign-users.

5. Smith's Models

Were we to assume that linguistic understanding presupposes non-linguistic understanding—that the use of words, for example, presupposes the acquisition of concepts—many of Smith's assertions would make good sense. Thus, insofar as acquiring a concept (a habit of thought or a habit of action) involves conceiving of the world in "a certain delimited way," it becomes apparent why, at least to some extent, as he suggests, it is necessary to build models (form concepts) "before building any artifact at all." Indeed, concepts themselves may be entertained as models that are

appropriate for thought or for action within suitable specific contexts. Building artifacts thus presupposes related models (concepts).

The two kinds of models that are of particular importance within computer science on Smith's account, therefore, are *specifications* (which outputs O are supposed to be derivable from which inputs I), on the one hand, and *programs* (which provide a sequence of instructions P for obtaining output of kind O from input of kind I), on the other. An example might be to generate an income tax return from inputs I concerning wages, salaries, and other income sources together with various types of medical, tax, and other deductions to calculate as output O the amount of additional taxes due or the refund owed to you. A program P that would achieve this objective would meet those specifications.

Not all programs that would achieve this objective are computer programs, insofar as the sets of instructions that the IRS provides can be implemented by any taxpaying citizen who can follow them; and not all computer programs that can achieve this objective are equally useful (in relation to their simplicity, economy, and efficiency). Some might actually be Rube Goldberg contraptions. Nevertheless, Smith has clearly identified two distinct kinds of models that are fundamental to computer science, which here may be characterized as follows:

Specifications (Model S): a description in a (usually formal) language that specifies the behavior desired from a system; and,

The Program (Model P): a set of instructions in a programming language indicating how that behavior is to be achieved.

Thus, when a program has been shown to meet its specifications—whether formally (by means of "program verification") or informally (by means of "program testing")—the programmer may be said to have "got the program right!" As Smith eloquently explains, however, even when Model P satisfies Model S, that only provides a "relative consistency proof" that one model satisfies the other: it does not guarantee what will happen should that program happen to be executed, because the specifications may or may not be suitably related to the world (Smith 1985/95, p. 465). Only when the right relationship has been established with the world has the programmer finally "got the right program!"

6. The Right-hand Side

Smith has contributed a valuable analysis of the role of models in computer science with respect to specifications and programs. When COMPUTER stands for Model P and MODEL stands for Model S with respect to Figure 1, therefore, the relationship between Model S and Model P appears to be straightforward. Indeed, a whole discipline, known as "model theory," is devoted to its analysis (Smith 1985/95, pp. 461–62). While this relationship is represented by the left side of Figure 1, Smith says that the right side of Figure 1 is problematical:

> The technical subject of "model theory", as I have already said, is the study of the relationship on the left. What about the relationship on the right? The answer, and one of the main points I hope you will take away from this discussion, is that, at this point in our intellectual history, we have no theory of this right-hand side relationship (Smith 1985/95, p. 462).

Thus, as Smith views the matter, we have "no theory" of the relationship (in the crucial case before us) between the REAL WORLD and MODEL as specifications.

In a previous critique of Smith's paper, I suggested that he had committed a blunder in overlooking the evident consideration that empirical science may be described as the search for an adequate *model of the world*, while philosophy of science, its complement, may be described as the search for an adequate *model of science* (Fetzer 1996, p. 252–54). After considering the matter further, however, I now believe that the situation is somewhat more complex in at least two respects, one of which I shall discuss here, the other in my concluding Section 9. The first concerns the development of specifications (Model S), while the other concerns attempts to ascertain whether or not we have "got the right system!"

A traditional distinction with the philosophy of science separates what has come to be called "the context of discovery" from "the context of justification." Thus, *the context of discovery* concerns the invention of hypotheses and theories, which characteristically have the status of guess or of conjectures; *the context of justification*, by comparison, concerns the appraisal of hypotheses and theories with respect to their relative degrees of evidential confirmation, which may include consideration of the clarity and precision of the language in which they are couched, their scope of

application for the purposes of explanation and prediction, the range of evidence in their support, and their relative degrees of simplicity, economy, or elegance (Hempel 1966, pp. 15–16).

The most important difference between them, for present purposes, may be that the invention of hypotheses and theories seems to be a *psychological process*, while the appraisal of hypotheses and theories seems to be a *logical procedure* (Fetzer 1993, Chapter 7). If the invention of specifications falls into the context of discovery, as it clearly does, then it is an activity that is not governed by algorithms that might be applied to problems of this kind. Thus, to this extent, Smith's contention that, "at this point in intellectual history, we have no theory of this right-hand side relationship," makes a certain point, but with an ambiguity that surely needs to be sorted out within computer science.

7. *Conceptualizations*

One way it might be explained is that the process of framing *specifications* presupposes a prior process of *conceptualization*. This stage of software development deserves greater emphasis than it generally tends to receive, perhaps for the reason Smith suggests ("we have no theory of this right-hand side relationship"). On the other hand, it is important to realize that conceptualization of the existence of a problem—grasping the existence of a problem that might be solved by the creation of a program P that takes values of the variable I as its input and that generates values of the variable O as its output—is a psychological process that involves creativity, intelligence, imagination, and experience.

Thus, to the extent to which the formulation of specifications presumes the existence of prior conceptualizations, the use of a (usually formal) language to describe the behavior that is desired of a system, as well as their underlying conceptualizations, may appropriately be held to represent problems to which programs represent possible solutions that might be subject to empirical test:

Conceptualization (Model C): classifying, categorizing, or otherwise subsuming a problem as an instance of a certain kind.

The formulation of specifications (Model S) thus presupposes the availability of conceptualizations (Model C) as a prior psychological achievement.

Without the conception of a problem to be solved, the formulation of specifications by means of any language would be a pointless activity highly unlikely to occur.

There appear to be at least two other kinds of models involved in the software production process that may not be sufficiently addressed by Smith's otherwise illuminating study. These involve the production of the program, on the one hand, and its execution by means of some machine, on the other. Although Smith emphasizes that programs are themselves models—indeed, they might be described as *models of solutions*—he does not attend to the role of programming languages themselves as models of another kind. And once we realize that higher-level languages, such as Pascal, LISP, and Prolog, stand for *virtual* rather than *physical* machines, it becomes plain that at least two more kinds of models have a role within the context of computer science.

Indeed, higher-level languages are related to physical machines by means of compilers and interpreters, which in some cases may not even exist, as in the case of the mini-language CORE (Marcotty and Ledgard 1986). CORE was introduced by Marcotty and Ledgard as a means for explaining the features that are characteristic of programming languages generally without encountering the complexities involved in discussions of Pascal, LISP, and so forth. In these kinds of cases, the programming language itself stands for a virtual machine as an abstract entity that may or may not be causally connected to any physical machine. They therefore exist as models of possible machines.

8. *Models and Languages*

It would be a mistake to suppose that programs as Models P are already sufficient to account for virtual machines represented by higher-level programming languages, especially because many—infinitely many—different programs can be formulated using the same high-level language. A high-level language thus functions as *a model of an abstract machine*, whereas a program is *a model of a potential solution* to a problem that might be executed by a machine of that kind. We might define these models as follows:

High-level programming languages (Model H): a set of rules that model behavior of an abstract (or virtual) machine;

where a virtual (or abstract) machine becomes a physical (or real) machine by virtue of the existence of some corresponding compilers or interpreters.

Thus, insofar as programs as texts are written in programming languages that model abstract machines, it remains the case that there may or may not be a suitable correspondence between the commands that occur within the language and the operations that are performed by some physical machine. The function of interpreters and compilers, therefore, is to create a causal mechanism whereby programs written in high-level languages can be executed by machines whose operations are causally affected by machine language, which typically consists of sequences of zeros and ones, as numerals that stand for patterns of switches set on-or-off, voltages that are high-or-low, and so on, depending upon the causal character of the specific system.

The great advantage of programming by means of high-level languages, of course, is that there is a one-many relationship between the commands that can be written in a high-level language and the counterpart operations that are performed by a machine executing them on the basis of their translation into machine language. Programming in machine language, indeed, is so difficult mentally that it is seldom encountered outside of the context of computer engineering, where there is no way to avoid it. So ordinarily the lowest-level language programmers use is assembly language, where there is a more or less one-to-one correspondence between commands and operations.

Both machine language and assembly language thus function as models of physical machines in a manner analogous to that in which higher-level languages function as models of abstract machines. The machines they represent may or may not exist as well, since there is no reason to deny the possibility of creating a mini-assembly language *á la* CORE for which there is no counterpart physical machine. For present purposes, however, we will assume that physical machines exist as causal counterparts to these low-level languages:

Lower-level programming languages (Model L): a set of rules that model the behavior of a physical (or causal) machine;

where a causal connection between higher-level languages (Model H) and lower-level languages (Model L) depends upon interpreters and compilers.

9. Embedding Relations

The relationship that obtains between programs (Model P), higher-level languages (Model H), and lower-level languages (Model L) can be represented by means of a simple diagram that schematizes their relations as follows:

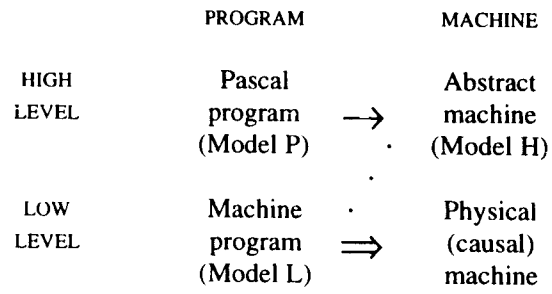

Figure 2. *Programs and Languages as Models*

where the single-arrow represents a possible relation between a program and the abstract machine represented by a high-level programming language and the double arrow represents an actual relation between a low-level program and the physical machine represented by its programming language. The series of three dots thus stands for the possible existence of compilers or interpreters that effect some causal connection between them.

Thus, strictly speaking, since programs written in Pascal, LISP, and other higher-level programming languages take for granted the existence of a corresponding virtual machine, their relationship may be described as that of one model (Model P) embedded within another (Model H). And whenever there exists a corresponding physical machine to which that higher-level language is related by means of a compiler or interpreter, then the degree of embeddedness ascends yet another step, since now one model (Model P) is embedded within another (Model H), which in turn is embedded within a third (Model L). And these are indeed embedded within the real world.

The conception of embedding may be envisioned as a logical relation or as a causal relation, depending upon the existence or the non-existence of a physical target machine. The construction of programs in CORE implies

the existence of virtual CORE machine, but not the existence of a physical target machine. A CORE program (Model P) is logically embedded in the CORE machine, but it is not causally embedded in a target machine, while a Pascal program (Model P) is logically embedded in Pascal (Model H) and also causally embedded (*via* interpreters, compilers, and assemblers) in machine language (Model L).[4]

10. The Embedding World

When programs are causaliy embedded (*via* interpreters, compilers, and assemblers) in physical target machines, then they have the capacity to causally interact with the world. This ordinarily occurs when they are components of *computer systems* as (typically complex) arrangements of computers, programs, and associated equipment that causally interact to bring about landings of real planes at real airports, the administration of real drugs, and the control of real radars, missiles and command systems. The openness of the right-hand side of Smith's Figure 1, therefore, serves as an important representation of the causal relation that obtains between computers, models, and the embedding world in cases of exactly this kind.

When Smith observes that even formal proofs that a program satisfies its specifications cannot guarantee that a system controlled by the program will do what it is supposed to do should that program be executed, the possible reasons why go beyond the consideration that those specifications may or may not stand in an appropriate relationship to the world. The conceptualization of the problem to be solved (Model C) may indeed be wrong and the specifications themselves (Model S) may be mistaken, as Smith remarks, but problems may also arise with the program (Model P), its embedding language (Model H) or its connections to the machine (Model L). Successful performance depends on the proper interaction of *all* the components of computer systems.

That the development and construction of a successful computer system is inherently complex may be underscored by the realization that different persons are typically responsible for different components of these systems. The conceptualization of the problem to be solved (Model C) may have originated with a product sponsor, but be translated into formal specifications (Model S) by a knowledge engineer and encoded into a program (Model P) by a team of programmers, which takes for granted that high-level languages (Model H) are suitably implemented by interpreters and compilers, and that low-level languages (Model L) are suitably

implemented by assemblers. And yet the success of the system still depends upon the interaction of its components.

The important role of empirical science within this context thus appears to apply distinctively to evaluating the performance of computer systems in the real world. Ideally, computer systems ought to be subjected to repeated tests under variable conditions in order to acquire information about their successful performance under those conditions. Data of this kind in the form of *relative frequencies* for successful performance can then be used to draw inferences about the *causal propensities* of these systems in the real world (Fetzer 1993, Chapter 6). In the real world, however, things seldom satisfy ideal standards. The consequential risks can be immense (Fetzer 1996).

While the conceptualization of specifications falls within the context of discovery, as we have found, the process of empirical test falls within the context of justification. Smith has properly delineated the role of models with respect to specifications and programs. Specifications are problems for which programs are potential solutions. But the role of models within computer science appears to be even more profound, where the use of languages as models of other kinds has to be appreciated to attain a sophisticated understanding of computer science. Indeed, even when we fully comprehend the left-hand-side relationship between models of one kind and models of another, we must also understand the right-hand-side relationship, if we are to surmise the role of science within computer science.

James H. Fetzer

University of Minnesota,
Duluth

NOTES

1. A referee has suggested that, rather than characterizing the *program* as a function that maps input onto output, it would be better to characterize the *specification* as the function, where a function is considered to be a certain kind of set input-output pairs. But a specification as a representation of a specific problem typically does not exist as a function but only as an invitation for the creation of a solution to that problem by means of a program.

2. A referee has proposed that the term 'computer' can be used in such a way as to imply that computers exist as causal systems, necessarily, where only Turing machines, for example, properly qualify as abstract entities. This approach, which seems to be the same as that of Smith (whose notion I am characterizing), ignores the difference between *actual computers* as physical systems and *virtual computers* as abstract things, which receives subsequent elaboration.

3. The same referee has also recommended that functions, as static sets of ordered pairs, should be differentiated from algorithms as dynamic entities that have the capacity to turn inputs into outputs. He concedes the possibility that algorithms might be taken as abstract entities, while programs are implementations of algorithms in forms that might be suitable for execution by causal machines, which is the perspective adopted here, as subsequent discussion will emphasize.

4. A referee has also observed that, while these distinctions are generally well-founded, dedicated systems provide a special case for which they do not obtain. In dedicated LISP machines, for example, high-level programs written in LISP simultaneously function as low-level assembly language programs, because the assembly language for dedicated LISP machines is LISP itself. This useful point also reflects the relativity of the distinction between "hardware" and "software."

REFERENCES

Fetzer, J. H. (1988), "Signs and Minds: An Introduction to the Theory of Semiotic Systems," in J. H. Fetzer, ed., *Aspects of Artificial Intelligence* (Dordrecht, The Netherlands: Kluwer Academic Publishers, 1988), pp. 133–61.

_____ (1990), *Artificial Intelligence: Its Scope and Limits* (Dordrecht, The Netherlands: Kluwer Academic Publishers, 1990).

_____ (1991/96), *Philosophy and Cognitive Science* (New York: Paragon House, 1991); 2nd ed'n. (Minneapolis, MN: Paragon House, 1996).

_____ (1993), *Philosophy of Science* (New York: Paragon House, 1993).

_____ (1994), "Mental Algorithms: Are Minds Computational Systems?", *Pragmatics & Cognition* 2 (1994), pp. 1–29.

_____ (1996), "Computer Reliability and Public Policy: Limits of Knowledge of Computer Based Systems," *Social Philosophy & Policy* 13 (1996), pp. 229–66. Reprinted in E. Paul, F. Miller, Jr., and J. Paul, eds., *Scientific Innovation, Philosophy and Public Policy* (New York: Cambridge University Press, 1996), pp. 229–66.

_____ (1998), "Thinking and Computing: Computers as Special Kinds of Signs," *Minds and Machines* 7 (1998), pp. 345–64.

Hempel, C. G. (1966), *Philosophy of Natural Science* (Englewood Cliffs, NJ: Prentice-Hall, 1966).

Marcotty, M. and H. Ledgard (1986), *Programming Language Landscape: Syntax/Semantics/Implementations*, 2nd ed'n. (Chicago: Science Research Associates, 1986).

Smith, B. C. (1985/95), "Limits of Correctness in Computers," *Technical Report SCLI: 85–35* (Stanford, CA: Center for the Study of Language and Information, 1985). Reprinted in D. Johnson and H. Nissenbaum, eds., *Computers, Ethics, & Social Values* (Englewood Cliffs, NJ: Prentice-Hall, 1995), pp. 456–69. References are to the reprinted version.