Copyright © 1985 Brian Cantwell Smith. Originally presented at a *Symposium on Unintentional Nuclear War* at the *Fifth Congress of the International Physicians for the Prevention Nuclear War*, Budapest, Hungary, June 28–July 1, 1985. The version delivered there was subsequently reprinted: (i) as Center for the Study of Language and Information Report CSLI–85–36, Stanford, California: Stanford University, Oct. 1985, 22 pp.; (ii) in D. Johnson & H. Nissenbaum (eds.), *Computers, Ethics & Social Values*, Englewood Cliffs, NJ: Prentice Hall, 456–69; (iii) in Colburn, T. R., Fetzer, J. H., & Rankin T. L. (eds.), *Program Verification*, Kluwer Academic Publishers, Dordrecht/Boston/London, 1993, pp. 275–93; and (iv) in Kling, R. (ed.), *Computerization and Controversy: Value Conflicts and Social Choices* (2nd Ed.), San Diego: Academic Press, pp. 810–25. Annotations added in 2014.

8 The Limits of Correctness

Abstract[†]

A0.5 Program verification is a technique in computer science that is used, in its own terms, to "prove programs correct." From its name, someone might easily conclude that a program that had been proven correct would never make any mistakes, or that it would always follow its designers intentions. In fact, however, what are called *proofs of correctness* are really proofs of the relative consistency between two formal specifications: one of the program, one of the model in terms of which the program is formulated. Part of assessing the correctness of a computer system, however, involves assessing the appropriateness of this model. Whereas standard semantical techniques are relevant to the program-model relationship, we do not currently have any theories of the further relationship between the model and the world in which the program is embedded.

In this paper I sketch the role of models in computer systems, comment on various properties of the model-world relationship, and suggest that [in the program verification context] the term 'correctness' should be changed to 'consistency.' In addition I argue that, since models cannot in general capture all the infinite richness of real-world domains, complete correctness is inherently unattainable, for people or for computers.

[†] This abstract was included in the versiom published as CSLI Technical Report 85–36; not in the SIGCAS newsletter version «check out others».

1 Introduction

On October 5, 1960, the American Ballistic Missile Early Warning System station at Thule, Greenland, indicated a large contingent of Soviet missiles headed towards the United States.[1] Fortunately, common sense prevailed at the informal threat-assessment conference that was immediately convened: international tensions were not particularly high at the time, the system had only recently been installed, Khrushchev[†] was in New York, and all in all a massive Soviet attack seemed very unlikely. And so no devastating counterattack was launched. What was the problem? The moon had risen, and was reflecting radar signals back to earth. Needless to say, this lunar reflection had not been predicted by the system's designers.

Over the last ten years,[‡] the United States Defense Department has spent many millions of dollars on a computer technology called "program verification"—a branch of computer science whose business, in its own terms, is to "prove programs correct." Program verification has been studied in theoretical computer science departments since a few seminal papers in the 1960s,[2] but it has only recently started to gain in public visibility, and to be applied to real world problems. General Electric, to consider just one example, has initiated verification

[1] Edmund Berkeley, The Computer Revolution, Doubleday, 1962, pp. 175–77, citing newspaper stories in the Manchester Guardian Weekly of Dec. 1, 1960, a UPI dispatch published in the Boston Traveler of Dec. 13, 1960, and an AP dispatch published in the New York Times on Dec 23, 1960.

[†] Nikita Khrushchev, Premier of the Soviet Union from 1958–64.

[‡] I.e., in the 1970s and early 1980s (the paper was presented in 1985).

[2] McCarthy, John, "A Basis for a Mathematical Theory of Computation," 1963, in P. Braffort and D. Hirschberg, eds., Computer Programming and Formal Systems, Amsterdam: North-Holland, 1967, pp. 33–70. Floyd, Robert, "Assigning Meaning to Programs," Proceedings of Symposia in Applied Mathematics 19, 1967 (also in F. T. Schwartz, ed., Mathematical Aspects of Computer Science, Providence: American Mathematical Society, 1967). Naur, P., "Proof of Algorithms by General Snapshots," BIT Vol. 6 No. 4, pp. 310–16, 1966.

projects in their own laboratories; they would like to prove that the programs used in their latest computer-controlled washing machines will not have any "bugs" (even a single serious one in a major product can destroy their profit margin).[3] Although it used to be that only the simplest programs could be "proven correct"—programs to put simple lists into order, to compute simple arithmetic functions, etc.—slow but steady progress has been made in extending the range of verification techniques. Recent papers have reported correctness proofs for somewhat more complex programs, including small operating systems, compilers, and other materiel of modern system A1 design.[4]

What, we do well to ask, does this new technology mean? How good are we at it? For example, if the 1960 warning system had been proven correct (which it was not), could we have avoided the problem with the moon? If it were possible to prove that programs being written to control automatic launch-on-warning systems were correct, would that mean there could not be a catastrophic accident? In systems now being proposed computers will make launching decisions in a matter of seconds, with no time for any human intervention (let alone for musings about Khrushchev's being in New York). Do the techniques of program verification hold enough promise so that, if these new systems could all be proven correct, we could all sleep more easily at night?

These are the questions I want to look at today. And my answer, to give away the punch-line, is *no*. For fundamental reasons—reasons that anyone can understand—there are inherent limitations to what can be proven about computers and computer programs. Although program verification is an important new technology, useful, like so many other things, in

[3] Albert Stevens, Raytheon BBN Technologies, Inc. [called "Bolt, Beranek and Newman" at the time], personal communication.

[4] See for example R. S. Boyer and Moore, J S., eds., The Correctness Problem in Computer Science, London: Academic Press, 1981.

its particular time and place, it should definitely not be called *verification*. Just because a program is "proven correct," in other words, you cannot be sure that it will do what you intend.

First some background.

2 General Issues in Program Verification

Computation is by now the most important enabling technology of nuclear weapons systems: it underlies virtually every aspect of the defense system, from the early warning systems, battle management and simulation systems, and systems for communication and control, to the intricate guidance systems that direct the missiles to their targets. It is difficult, in assessing the chances of an accidental nuclear war, to imagine a more important question to ask than whether these pervasive computer systems will or do work correctly.

Because the subject is so large, however, I want to focus on just one aspect of computers relevant to their correctness: the use of *models* in the construction, use, and analysis of computer systems. I have chosen to look at modeling because I think it exerts the most profound and, in the end, most important influence on the systems we build. But it is only one of an enormous number of important questions. First, therefore—in order to unsettle you a little—let me just hint at some of the equally important issues I will not address:

1. **Complexity:** At the current state of the art, only very simple programs can be proven correct. Although it is terribly misleading to assume that either the complexity or power of a computer program is a linear function of length, some rough numbers are illustrative. The simplest possible arithmetic programs are measured in tens of lines; the current state of the verification art extends only to programs of up to several hundred. It is estimated that the systems proposed in the Strategic Defense Initiative (Stars Wars), in contrast, will

require at least 10,000,000 [ten billion] lines of code.[5] By analogy, compare the difference between resolving a two-person dispute and settling the political problems of the Middle East. There is no a priori reason to believe that strategies successful at one level will scale to the other.

2. **Human interaction:** Not much can be "proven," let alone specified formally, about actual human behavior. The sorts of programs that have so far been proven correct, therefore, do not include much substantial human interaction. On the other hand, as the moonrise example indicates, it is often crucial to allow enough human intervention to enable people to override system mistakes. System designers, therefore, are faced with a very real dilemma: should they rule out substantive human intervention, in order to develop more confidence in how their systems will perform; or should they include it, so that costly errors can be avoided or at least repaired? The Three Mile Island incident[†] is a trenchant example of just how serious this tradeoff can get: the system design provided for considerable human intervention, but then the operators failed to act "appropriately." Which strategy leads to the more important kind of correctness?

[5] Fletcher, James C., study chairman, and McMillan, Brockway, panel chairman, Report of the Study on Eliminating the Threat Posed by Nuclear Ballistic Missiles (U), Vol. 5, Battle Management, Communications, and Data Processing (U), U. S. Department of Defense, February 1984.

[†] Five years before this paper was written, on March 28, 1979, a nuclear power plant on Three Mile Island near Harrisburg, Pennsylvania, suffered a partial nuclear meltdown, resulting in the release of small amounts of radioactive iodine and radioactive gas into the environment. The reactor was ultimately brought under control, but to this day the accident remains the worst in the history of U.S. nuclear power industry. Though technically referring to the island itself, the term "Three Mile Island" still effectively serves as a proper name for the singular accident.

A standard way out of this dilemma is to specify the behavior of the system *relative to the actions of its operators*. But as we will see below, this strategy pressures the designers to specify the system totally in terms of internal actions, not external effects. So you end up proving only that the system will *behave in the way that it will behave* (i.e., it will raise this line level 3 volts), not *do what you want it to do* (i.e., launch a missile only if the attack is real). Unfortunately, the latter is clearly what is important. Systems comprising computers and people must function properly as integrated systems; nothing is gained by showing that one cog in a misshapen wheel is a very nice cog indeed.

Furthermore, large computer systems are dynamic, constantly changing, embedded in complex social settings. Another famous "mistake" in the American defense system occurred when a human operator mistakenly mounted a training tape, containing a simulation of a full-scale Soviet attack, onto a computer that, just by chance, was automatically pulled into service when the primary machine ran into a problem. For some tense moments the simulation data were taken to be the real thing.[6] What does it mean to install a "correct" module into a complex social flux?

3. **Levels of Failure:** Complex computer systems must work at many different levels. It follows that they can fail at many different levels too. By analogy, consider the many different ways a hospital could fail. First, the beams used to frame it might collapse. Or they might perform flawlessly, but the operating room door

[6] See, for example, the Hart-Goldwater report to the Committee on Armed Services of the U.S. Senate: "Recent False Alerts from the Nation's Missile Attack Warning System" (Washington, D.C.: U.S. Government Printing Office, Oct. 9, 1980); Physicians for Social Responsibility, Newsletter, "Accidental Nuclear War," (Winter 1982), p. 1.

might be too small to let in a hospital bed (in which case you would blame the architects, not the lumber or steel company). Or the operating room might be fine, but the hospital might be located in the middle of the woods, where no one could get to it (in which case you would blame the planners). Or, to take a different example, consider how a letter could fail. It might be so torn or soiled that it could not be read. Or it might look beautiful, but be full of spelling mistakes. Or it might have perfect grammar, but disastrous contents.

Computer systems are the same: they can be "correct" at one level—say, in terms of hardware—but fail at another (i.e., the systems built on top of the hardware can do the wrong thing even if the chips are fine). Sometimes, when people talk about computers failing, they seem to think that only the hardware needs to work. And hardware does from time to time fail, causing machines to come to a halt, or yielding errant behavior (as for example when a faulty chip in another American early warning system sputtered random digits into a signal interpreted as indicating how many Soviet missiles had been sighted, again causing a false alert[7]). And the connections between the computers and the world can break; when the moonrise problem was first recognized, an attempt to override it failed because an iceberg had accidentally cut an undersea telephone cable.[8]

But the more important point is that, in order to be reliable, a system must be correct, or anyway reliable, *at every relevant level*; the hardware is just the starting place (and by far the easiest, at that). Unfortunately, however, we do not even know what all the relevant

[7] Ibid.

[8] Berkeley, op. cit. See also Daniel Ford's two-part article "The Button," *New Yorker*, April 1, 1985, p. 43, and April 8, 1985, p. 49, excerpted from Ford, Daniel, *The Button*, New York: Simon and Schuster, 1985.

levels are. So-called "fault-tolerant" computers, for example, are particularly good at coping with hardware failures, but the software that runs on them is not thereby improved.[9]

4. **Correctness and Intention:** What does *correct* mean, anyway? Suppose the people want peace, and the President thinks that means having a strong defense, and the Defense Department thinks that means having nuclear weapons systems, and the weapons designers request control systems to monitor radar signals, and the computer companies are asked to respond to six particular kinds of radar pattern, and the engineers are told to build signal amplifiers with certain circuit characteristics, and the technician is told to write a program to respond to the difference between a two-volt and a four-volt signal on a particular incoming wire. If being correct means *doing what was intended*, whose intent matters? The technician's? Or what, with twenty years of historical detachment, we would say *should have been intended?*

With a little thought any of you could extend this list yourself. And none of these issues even touch on the intricate technical problems that arise in building the mathematical models of software and systems used in the so-called "correctness" proofs. But, as I said, I want to focus on what I take to be the most important issue underlying all of these concerns: the pervasive use of *models*. Models are ubiquitous not only in computer science but also in human thinking and language; their very familiarity makes them hard to appreciate. So we will start simply, looking at modeling on its own, and come back to correctness in a moment.

[9] Developing software for fault-tolerant systems is in fact an extremely tricky business.

3 The Permeating Use of Models

When you design and build a computer system, you first formulate a model of the problem you want it to solve, and then A3 construct the computer program in its terms. For example, if you were to design a medical system to administer drug therapy, you would need to model a variety of things: the patient, the drug, the absorption rate, the desired balance between therapy and toxicity, and so on and so forth. The absorption rate might be modeled as a number proportional to the patient's weight, or proportional to body surface area, or as some more complex function of weight, age, and sex.

Similarly, computers that control traffic lights are based on some model of traffic—of how long it takes to drive across the intersection, of how much metal cars contain (the signal change mechanisms are triggered by wires buried under each street). Bicyclists, as it happens, often have problems with automatic traffic lights, because bicycles do not exactly fit the model: they do not contain enough iron to trigger the metal detectors. I also once saw a tractor get into trouble because it could not move as fast as the system "thought" it would: the cross-light went green when the tractor was only half-way through the intersection.

To build a model is to conceive of the world in a certain delimited way. To some extent you must build models before A4 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 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. In fact that is really what computers are (and how they differ from other machines): they run by manipulating representations, and representations are always formulated in terms of models. This can all be summarized in a slogan: *no computation without representation*. A5

The models, on which the representations are based, come in all shapes and sizes. Balsa models of cars and airplanes, for example, are used to study air friction and lift. Blueprints can be viewed as models of buildings; musical scores as models of a symphony. But models can also be abstract. Mathematical models, in particular, are so widely used that it is hard to think of anything that they have not been used for: from whole social and economic systems, to personality traits in teenagers, to genetic structures, to the mass and charge of sub-atomic particles. These models, furthermore, permeate all discussion and communication. Every expression of language can be viewed as resting implicitly on some model of the world.

What is important for our purposes is that every model deals with its subject matter *at some particular level of abstraction*, paying attention to certain details, throwing away others, grouping together similar aspects into common categories, and so forth. So the drug model mentioned above would probably pay attention to the patients' weights, but ignore their tastes in music. Mathematical models of traffic typically ignore the temperaments of taxi drivers. Sometimes what is ignored is [considered to be] at too "low" a level, sometimes too "high"; it depends on the purposes for which the model is being used. So a hospital blueprint would pay attention to the structure and connection of its beams, but not to the arrangements of proteins in the wood the beams are made of, nor to the efficacy of the resulting operating room.

Models *have to* ignore things exactly because they view the world at a level of abstraction ('abstraction' is from the Latin *abstrahere*, 'to pull or draw away'). And it is good that they do: otherwise they would drown in the infinite richness of the embedding world. Though this is not the place for metaphysics, it would not be too much to say that every act of concep- A6 tualization, analysis, categorization, does a certain amount of violence to its subject matter, in order to get at the underlying regularities that group things together. If you do not commit that act of violence—do not ignore some of what is going on—you would become so hypersensitive and so overcome with complexity that you would be unable to act.

To capture all this in a word, I will say that models are inherently *partial*. All thinking, and all computation, are similarly partial. Furthermore—and this is the important point—thinking and computation *have* to be partial: that's how they A7 are able to work.

4 Full-blooded Action

Something that is not partial, however, is action. When you reach out your hand and grasp a plow, it is the real field you are digging up, not your model of it. Models, in other words, may be abstract, and thinking may be abstract, and some aspects of computation may be abstract, but action is not. To actually build a hospital, to clench the steering wheel and drive through the intersection, or to inject a drug into a person's body, is to act in the full-blooded world, not in a partial or distilled model of it.

This difference between action and modeling is extraordinarily important. Even if your every thought is formulated in the terms of some model, to act is to take leave of the model and participate in the whole, rich, infinitely variegated world. A8 For this reason, among others, action plays a crucial role, especially in the human case, in grounding the more abstract processes of modeling or conceptualization. One form that grounding can take, which computer systems can already take advantage of, is to provide feedback on how well the modeling is going. For example, if an industrial robot develops an internal three-dimensional representation of a wheel assembly passing by on a conveyor belt, and then guides its arm towards that object and tries to pick it up, it can use video systems or force sensors to see how well the model corresponded to what was actually the case. The world does not care about the model: the claws will settle on the wheel just in case the actualities mesh.

Feedback is a special case of a very general phenomenon: you often learn, when you do act, just how good or bad your conceptual model was. You learn, that is, if you have adequate sensory apparatus, the capacity to assess the sensed experience, A9 the inner resources to revise and reconceptualize, and the luxury of recovering from minor mistakes and failures.

5 Computers and Models

What does all this have to do with computers, and with correctness? The point is that computers, like us, participate in the real world: they take real actions. One of the most important facts about computers, to put this another way, is that we plug them in. 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 we know only too well—control real radars, missiles, and command systems. Like us, in other words, although they base their actions on models, they have consequence in a world that inevitably transcends the partiality of their enabling models. Like us, in other words, and unlike the objects of mathematics, they are challenged by the inexorable conflict between partial but tractable models and actual but infinite world.

And, to make the only too obvious point: we in general have no guarantee that the models are right—indeed we have no guarantee about much of anything about the relationship between model and world. As we will see, current notions of "correctness" do not even address this fundamental question.

✦ ✦ ✦

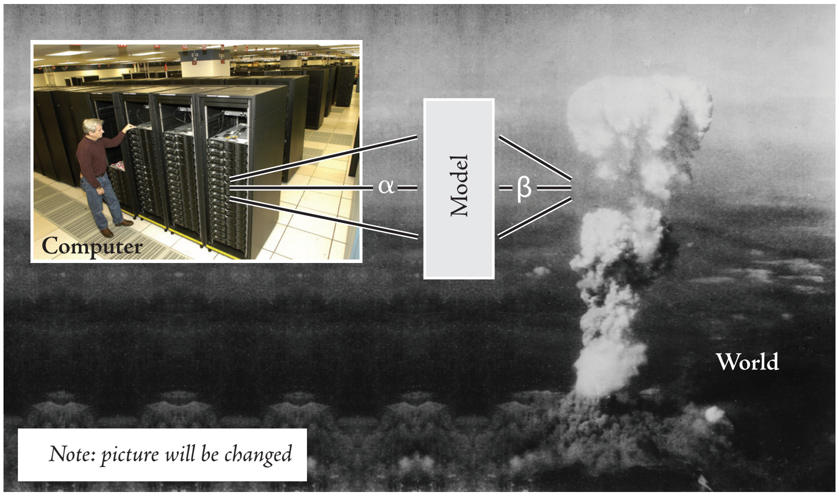

Figure 1 — Computers, Models, and the Embedding World

In philosophy and logic, as it happens, there is a very precise mathematical theory called "model theory." You might think that it would be a theory about what models are, what they are good for, how they correspond to the worlds they are models of, and so forth. You might even hope this was true, for the following reason: a great deal of theoretical computer science, and all of the work in program verification and correctness, historically derives from this model-theoretic tradition, and depends on its techniques. Unfortunately, however, model theory does not address the model-world relationship at all. Rather, what model theory does is to tell you how your descriptions, representations, and programs *correspond to your model*.

The situation, in other words, is roughly as depicted in figure 1. 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 the right. Mediating between the two is the inevitable model, serving as an idealized or pre-conceptualized 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: it is the world, that is, as the system sees it (though not, of course, as it necessarily is).

The technical subject of "model theory," as I have already said, is a study of the relationship on the left [labeled α]. What about relationship on the right [labeled β]? The answer, and one of the main points I hope you will take away from this discussion, is that, at this point in intellectual history, *we have no theory of this right-hand side relationship*.

There are lots of reasons for this [lack], some very complex. For one thing, most of our currently accepted formal techniques were developed during the first half of this century to deal with mathematics and physics. Mathematics is unique, with respect to models, because (at least to a first level of approximation) its subject matter *is* the world of models and abstract structures, and therefore the model-world relationship A10 is relatively unproblematic. The situation in physics is more complex, of course, as is the relationship between mathematics and physics. How apparently pure mathematical structures can be so successfully used to model the material substrate of the universe is a question that has exercised physical scientists for centuries. But the point is that, whether A11 or not one believes that the best physical models do more justice and therefore less violence to the world than do models in so-called "higher-level" disciplines like sociology or economics, formal techniques do not themselves address the question of [the model's] adequacy.

Another reason we do not have a theory of the right-hand side is that there is very little agreement on what such a theory would look like. In fact all kinds of question arise when one studies the model-world relationship explicitly, about whether it can be treated formally at all, whether it can be treated rigorously, even if not formally (and what the relationship is between those two), about whether any theory will be more than usually infected with the prejudices and preconceptions of the theorist, and so forth. The investigation quickly leads to foundational A12 questions in mathematics, philosophy, and language, as well as computer science. But none of what one learns in any way lessens its ultimate importance. In the end, any adequate theory of action, and, consequently, any adequate theory of correctness, will have to take the model-world relationship into account.

6 Correctness and Relative Consistency

Let's get back, then, to computers, and to correctness. As I mentioned earlier, the word 'correct' is already problematic, especially as it relates to underlying intention. Is a program correct when it does what we have instructed it to do? or what we wanted it to do? or what history would dispassionately say it should have done? Analyzing what correctness *should* mean is too complex a topic to take up directly. What I want to do, in the time remaining, is to describe what sorts of correctness we are presently capable of analyzing.

In order to understand this, we need to understand one more thing about building computer systems. I have already said that, when you design a computer system, you first develop a model of the world, as indicated in Figure 1. But you don't, in general, ever get to hold the model in your hand: computer systems, in general, are based on models that are purely abstract. Rather, if you are interested in proving your program "correct," you develop two concrete things, structured in terms of the abstract underlying model (although these are listed here in logical order, the program is very often written first):

1. A **specification**: a formal description in some standard formal language, specified in terms of the model, in which the desired behavior is described; and

2. The **program**: a set of instructions and representations, also formulated in the terms of the model, which the A13 computer uses as the basis for its actions.

How do these two differ? In various ways, of which one is particularly important. The program has to say *how the behavior is to be achieved*, typically in a step-by-step fashion (and often in excruciating detail). The specification, however, is less constrained: all it has to do is to specify *what proper behavior would be*, independent of how it is accomplished. For example, a specification for a milk delivery system might simply be: "Make one milk delivery at each store, driving the shortest possible distance in total." That's just a description of *what* has to happen. The program, on the other hand, would have the much more difficult job of saying *how* this was to be accomplished. It might be phrased as follows: "Drive four blocks north, turn right, stop at Gregory's Grocery Store on the corner, drop off the milk, then drive 17 blocks north-east..." Specifications, to use some of the jargon of the field, are essentially *declarative*; they are like indicative sentences or claims. A14 Programs, on the other hand, are *procedural*: they must contain instructions that lead to a determinate sequence of actions.

What, then, is a proof of correctness? It is a proof that any system that *obeys the program* will *satisfy the specification*.

There are, as is probably quite evident, two kinds of problems here. The first, often acknowledged, is that the correctness proof is in reality only a proof that two characterizations of something are compatible. When the two differ—i.e., when you try to prove correctness and fail—there is no more reason to believe that the first (the specification) is any more correct than the second (the program). As a matter of technical practice, specifications tend to be extraordinarily complex formal descriptions, just as subject to bugs and design errors and so forth as programs. In fact they are very much *like* programs, as this introduction should suggest. So what almost always happens, when you write a specification and a program, and try to show that they are compatible, is that you have to adjust *both* of them in order to get them to converge.

For example, suppose you write a program to factor a number C, producing two answers A and B. Your specification might be:

Given number C, produce numbers A and B such that A×B=C

This is a specification, not a program, because it does not tell you how to come up with A and B; all it say is what properties A and B should have. In particular, suppose I say: "OK, C A15 is 5,332,114; what are A and B? Staring at the specification just given will not help you to come up with the answer. Suppose, A16 on the other hand, given this specification, that you then write a program—say, by successively trying pairs of numbers until you find two that work. Suppose further that you then set out to prove that your program meets your specification. And, finally, suppose that this proof can be constructed (I will not go into details here; I trust you can imagine that such a proof could be constructed). With all three things in hand—program, specification, and proof—you might think you were done.

In fact, however, things are rarely that simple, as even this simple example can show. In particular, suppose, after doing all this work, that you try your program out on some simple examples, confident that it must work because you have a proof of its correctness. You randomly give it 14 as an input, expecting 2 and 7. But in fact it gives you the answers A=1 and B=14. In fact, you realize upon further examination, it will *always* give back A=1 and B=C. It does this, *even though you have a proof of its being correct*, because you did not make your specification meet your intentions. You wanted both A and B to be *different* from C (and also different from 1), but you forgot to say that. In this case you have to modify both the program and the specification. A plausible new version of the latter would be:

> *Given number* C, *produce numbers* A *and* B *such that* $A \neq 1$ *and* $B \neq 1$ *and* $A \times B = C$.

And so on and so forth; the point, I take it, is obvious. If the next version of the program, given C=14, produces A=-1 and B=-14, you would once again have met your new specification, but still failed to meet your intention. Writing "good" specifications—which is to say, writing specifications that capture your intention—is hard.

It should be apparent, nonetheless, that developing even straightforward proofs of "correctness" is nonetheless very useful. It typically forces you to delineate, very explicitly and completely, the model on which both program and specification are based. A great many of the simple bugs that occur in programs, of which the problem of producing 1 and 14 was an example, arise from sloppiness and unclarity about the model. A17 Such bugs are not identified, per se, by the proof, but they are often unearthed in the attempt to prove [the equivalence]. And of course there is nothing wrong with this practice; anything that helps to eradicate errors and increase confidence is to be applauded. The point, rather, is to show exactly what these proofs consist in.

In particular, as the discussion has shown, when you show that a program meets its specifications, all you have done is to show that two formal descriptions, slightly different in character, are compatible. This is why I think it is somewhere between misleading and immoral for computer scientists to call this "correctness." What is called a proof of correctness is really a proof of the compatibility or consistency between two formal objects of an extremely similar sort: program and specification. As a community, we computer scientists should call this **relative consistency**, and drop the word 'correctness' completely.

What proofs of relative consistency ignore is the second problem intimated earlier. Nothing in the so-called program verification process per se deals with the right-hand side relationship: the relationship between the model and the world. But, as is clear, it is over inadequacies on the right hand side—inadequacies, that is, in the models in terms of which the programs and specifications are written—that systems so commonly fail.

The problem with the moonrise, for example, was a problem of this second sort. The difficulty was not that the program failed, in terms of the model. The problem, rather, was that the model was overly simplistic; it *did not correspond to what was the case in the world*. Or, to put it more carefully, since all models fail to correspond to the world in indefinitely many ways, as we have already said, it did not correspond to what was the case *in a crucial and relevant way*. In other words, to answer one of our original questions, even if a formal specification had been written for the 1960 warning system, and a proof of correctness generated, there is no reason to believe that potential difficulties with the moon would have emerged. A18

You might think that the designers were sloppy; that they would have thought of the moon if they had been more careful. But it turns out to be extremely difficult to develop realistic models of any but the most artificial situations, and to assess how adequate these models are. To see just how hard it can be, think back on the case of General Electric, and imagine writing appliance specifications, this time for a refrigerator. To give the example some force, imagine that you are contracting the manufacture of the refrigerator out to an independent supplier, and that you want to put a specification into the contract that is sufficiently precise to guarantee that you will be happy with anything that the supplier delivers that meets the contract.

Your first version might be quite simple—say, that the requisitioned device should maintain an internal temperature of between 3 and 6 degrees Centigrade; not use more than 200 A19 watts of electricity; cost less than $100 to manufacture; have an internal volume of half a cubic meter; and so on and so forth. But of course there are hundreds of other properties that you implicitly rely on: it should, presumably, be structurally sound: you would not be happy with a deliciously cool plastic bag. It should not weigh more than a ton, or emit loud noises. And it should not fling projectiles out at high speed when the door is opened. In general, it is impossible, when writing specifications, to include *everything* that you want: legal contracts, and other humanly interpretable specifications, are always stated within a background of commonsense, to cover the myriad unstated and unstatable assumptions assumed to hold in force. (Current computer, alas, have no common sense, as the cartoonists know so well.)

So it is hard to make sure that everything that meets your specification will really be a refrigerator; it is also hard to make sure that your requirements do not rule out perfectly good refrigerators. Suppose for example a customer plugs a toaster in, puts it inside the refrigerator, and complains that the object they received does not meet the temperature specification, and must therefore not be a refrigerator. Or suppose they try to run it upside down. Or complains that it does not work in outer space, even though you did not explicitly specify that it would only work within the earth's atmosphere. Or spins it at 10,000 rpm. Or even just unplugs it. In each case you would say that the problem lies not with the refrigerator but with the use. But how is *use* to be specified? The point is that, as well as modeling the artifact itself, you have to model the relevant part of the world in which it will be embedded. It follows that the model of a refrigerator as a device that always maintains an internal temperature of between 3 and 6 degrees is too strict to cover all possible situations. One could try to model A20 what appropriate use would be, though specifications do not, ordinarily, even try to identify all the relevant circumstantial factors. As well as there being a background set of constraints with respect to which a model is formulated, there is also a background set of assumptions on which a specification is allowed at any point to rely.

7 The Limits of Correctness

It's time to summarize what we have said so far. The first challenge to developing a perfectly "correct" computer system stems from the sheer complexity of real-world tasks. We mentioned at the outset various factors that contribute to this complexity: human interaction, unpredictable factors of setting, hardware problems, difficulties in identifying salient levels of abstraction, etc. Nor is this complexity of only theoretical concern. A December 1984 report of the American Defense Science Board Task Force on "Military Applications of New-Generation Computing Technologies" identifies the following gap between current laboratory demonstrations and what will be required for successful military applications—applications they call "Real World; Life or Death." In their esti- A21 mation the military now[†] needs (and, so far as one can tell, expects to produce) an increase in the power of computer systems of nine decimal orders of magnitude, accounting for both speed and amount of information to be processed. That is a 1,000,000,000-fold [one billion-fold] increase over current research systems, equivalent to the difference between a full century of the entire New York metropolitan area, compared to one day in the life of a hamlet of one hundred people. And remember that even current systems are already several orders of magnitude more complex that those for which we can currently develop proofs of relative consistency.

But sheer complexity has not been our primary subject

[†] I.e., in the mid-1980s.

matter. The second challenge to computational correctness, more serious, comes from the problem of formulating or specifying an appropriate model. Except in the most highly artificial or constrained domains, modeling an embedding situation is an approximate, not a complete, endeavour. It has the best hopes of even partial success in what Winograd has called "systematic domains": areas where the relevant stock A22 of objects, properties, and relationships are most clearly and regularly predefined. Thus bacteria, or warehouse inventories, or even flight paths of airplanes coming into airports, are relatively systematic domains, at least compared to conflict negotiations, any situations involving intentional human agency, learning and instruction, and so forth. The systems that land airplanes are hybrids—combinations of computers and people—exactly because the unforeseeable happens, because what happens is in part the result of human action, requiring human interpretation. Although it is impressive how well the phone companies can model telephone connections, lines, and even develop statistical models of telephone use, at a certain level of abstraction, it would nevertheless be impossible to model the *content* of the telephone conversations themselves.

Third, and finally, is the question of what one does about these first two facts. It is because of the answer to this last question that I have talked, so far, somewhat interchangeably about people and computers. With respect to the ultimate limits of models and conceptualization, both people and computers are restrained by the same truths. If the world is infinitely rich and variegated, then no prior conceptualization of it, nor any abstraction, will ever do it full justice. That is OK—or at least we might as well say that it is OK, since that is the world we have got. What matters is that we never forget about that richness—that we not think, with misplaced optimism, that machines might magically have access to a kind of "correctness" to which people cannot even aspire.

It is time, to put this another way, that we change the traditional terms of the debate. The question is not whether *machines* can do things, as if, in the background, lies the implicit assumption that the object of comparison is *people*. Plans to build automated systems capable of making a "decision," in a matter of seconds, to annihilate Europe, say, should make you uneasy; requiring a *person* to make the same decision in a matter of the same few seconds should make you uneasy too, and for very similar reasons. The problem is that there is simply no way that reasoning of any sort can do justice to the inevitable complexity of the situation, *because of what reasoning is*. Reasoning is based on partial models. Which means it cannot be guaranteed to be correct. Which means, to suggest just one possible strategy for action, that we might try, in our treaty negotiations, to find mechanisms to *slow our weapons systems down*.

It is striking to realize, once the comparison between machines and people is raised explicitly, that we do not typically expect "correctness" for people in anything like the form that that we presume it for computers. In fact quite the opposite, and in a revealing way. Imagine, in some by-gone era, sending a soldier off to war, and giving him (it would surely have been a "him") final instructions. "Obey your commander; help your fellow-soldier," you might say, "and above all do your country honour." What is striking about this is that it is considered not just a weakness, but a punishable weakness—a breach of morality—to obey instructions *blindly* (in fact, and for relevant reasons, you generally *can't* follow instructions blindly; they have to be interpreted to the situation at hand). You are subject to court martial, for example, if you violate fundamental moral principles, such as murdering women and children, even if following strict orders.

In the human case, in other words, our social and moral systems seem to have built in in an acceptance of the uncertainties and limitations inherent in the model-world relationship [relation β in figure 1]. We *know* that the assumptions and preconceptions built into instructions will sometimes fail, and we *know* that instructions are always incomplete; we exactly rely on judgment, responsibility, consciousness, and so forth, to carry someone through those situations—all situations, in fact—where model and world part company. In fact we never talk about people, in terms of their overall personality, being *correct;* we talk about people their being *reliable,* a much more substantial term. It is individual actions, fully situated in a particular setting, that are correct or incorrect, not people in general, or systems. What leads to the highest number of correct human actions is a person's being reliable, experienced, capable of good judgment, etc.

There are two possible morals here, for computers. The first has to do with the notion of experience. In point of fact, program verification is not the only, or even the most common, method of obtaining assurance that a computer system will do the right thing. In the real world, programs are usually judged acceptable, and are typically accepted into use, not because we prove them "correct," but because they have shown themselves relatively reliable in their destined situations for some substantial period of time. And, as part of this experience, we *expect* them to fail: there always has to be room for failure. Certainly no one would ever accept a program without this *in situ* testing: a proof of correctness is at best added insurance, not a replacement, for real-life experience. Unfortunately, for the ten million lines of code that is supposed to control and coordinate the Star Wars Defense System, there will never, God willing, be an *in situ* test.

One answer, of course, if genuine testing is impossible, is to run a *simulation* of the real situation. But simulation, as our diagram should make clear, *tests only the left-hand side re-*

lationship [α in figure 1]. Simulations are defined in terms of models; they do not test the relationship between the model and the world. That is exactly why simulations and tests can never replace embedding a program in the real world. All the war games we hear about, and hypothetical military scenarios, and electronic battlefield simulators, and so forth, are all based on exactly the kinds of models we have been talking about all along. In fact the subject of simulation, worthy of a whole analysis on its own, is really just our whole subject welling up all over again.

I said earlier that there were two morals to be drawn, for the computer, from the fact that we ask people to be reliable, not to be correct. The second moral is for those who, when confronted with the fact that genuine or adequate experience cannot be had, would say "Oh, well, let's build responsibility and morality into computers—if people can have it, there is no reason why machines can't have it too." Now I will not argue that this is inherently impossible, in some metaphysical or ultimate philosophical sense, but a few short comments are in A23 order. First, from the fact that humans sometimes *are* responsible, it does not follow that we know what responsibility is: from tacit skills no explicit model is necessarily forthcoming. We simply do not know what aspects of the human condition underlie the modest levels of responsibility to which we sometimes rise. And second, with respect to the goal of building computers with even human levels of full reliability and responsibility, I can state with surety that the present state of artificial intelligence is about as far from this as mosquitoes are from flying to the moon.

But there are deeper morals even than these. The point is that even if we *could* make computers reliable, they still wouldn't necessarily always do the correct thing. *People* are not provably "correct," either; that's why we hope they are responsible, and surely one of the major ethical facts is that correctness and responsibility do not coincide. Even if, in another 1,000 years, someone were to devise a genuinely responsible A24 computer system, there is no reason to suppose that it would achieve "perfect correctness" either, in the sense of never doing anything wrong. This isn't a failure, in the sense of a performance limitation; it stems from the deeper fact that models must be abstract, in order to be useful. The lesson to A25 be learned from the violence inherent in the model-world relationship, in other words, is that there is an *inherent* conflict between the power of analysis and conceptualization, on the one hand, and sensitivity to the infinite richness, on the other.

But perhaps this is an overly abstract way to put it. Perhaps, instead, we should just remember that there will always be another moonrise.

Annotations[1]

A0.5 ·:… «…Write a general intro. A very “public” paper, written on a long plane flight from San Francisco to Budapest; more read than everything else I’ve written combined. INcluded here because it exemplifies, in concrete, real-world, publicly-accessible way, many of the issues with which the technical work is concerned: deferential relations to the real world external to the computer; issues of registration and intent, etc. …»

A1 ·3/0/-3:-1 It is significant that the programs of any length for which correctness has been proved (including the ones listed here: operating systems and compilers) have “internal” subject matters—i.e., they are programs whose task domains are, reflexively, programs and computing. Not only is programming a paradigmatic “systematic domain,” in the sense discussed at ……,[1] but formalising the progamming domain is already part of the task of proving any program correct, and so dealing with programs about programs is the simplest possible case, with the possible exception of pure mathematics.

Needless to say, nuclear war is not an internal subject matter.

A2 ·7/-1:8/0 Questions about the form of and relations among theories of one and the same system at multiple levels of analysis permeate science and philosophy of science (e.g., in the case of the mind, about what sorts of regularities hold of the human mind/brain at social, psychological, neurological, biochemical, chemical, and physical levels). One of the most powerful and ubuiquitous techniques in all of computing is the ability to “implement” one system on top of another, engendering the same set of questions (“What holds true at what level?”) for any computer system of greater than trivial complexity. Not only do we still have no theory of how to allocate responsibility and/or theoretical attention across these levels; we do not even have conceptual machinery to help us identify the salient levels. «…References to other annotations on this point…»

One particularly challenging point ties into the semantical analyses that permeate the discussion of 3Lisp and reflection in Part B: the inclusion of declarative or representation semantics (φ) along with procedural or behavioral (ψ). One of the reasons representation is challenging is that representation does not “cross implementation

†References are in the form page/paragraph/line; with ranges (of any type) indicated as x:y. For details see the explanation on p.·:…

[1] See also annotation A…

levels," in the sense that if a system representing the traffic patterns of itinerant farm workers is implemented in C++, then it may be more natural to say that a given C++ variable *represents an element in the data base* rather than representing *what the data base element in turn represents*—e.g., a particular farm worker. «…Refs to annotations where crossing implementation boundaries is discussed…» Or for a more philosophically familiar example, imagine a first-order logic representation of the English sentence "A fire has started in engine room #2." The natural semantical analysis of the variables employed in the logical axiomatisation would be likely to take their referential domain to consist of *English sentences themselves*, rather than what those English sentences are about—viz, fires.

Put it this way: issues of 'use' and 'mention' in computer systems are stupefyingly complex. Cf. chapter 12 ("The Correspondence Continuum").

A3 ·9/1/1:2 The way this is phrased ("you first formulate a model of the problem you want it to solve") suggests that the model formulation must be witting and explicit—which is by no means always the case. The point is merely that any program must be framed, implicitly or explicitly, with respect to what I am here calling a "model" of its subject matter—i.e., with respect to a "take" or "way of conceptualising" its task domain. As noted in «…», I would today use the terminology of 'registration' instead of that of models—which I believe would make the analysis more accute, though perhaps less widely accessible. Thus I would tend to write the first sentence along something like the following lines: that when you design and build a computer system, you do so "in terms of an (implicit or explicit) registration of the problem you want to solve."

Cf. also the discussion at …, and the immediately following annotation (A…); also «…» in Volume II, on how such registrations need not be (what I there call) *conceptual*.

A4 ·9/-1/1:2 When I say "[t]o build a model is to conceive of the world in a certain delimited way" I am effectively defining my use of the term 'model.' That is, the sentence should be read as meaning roughly that I employ the phrase "use a model" as shorthand for "conceive of the world in a certain delimited way." Cf. also the last sentence of 10/1 ("[e]very expression of language can be viewed as resting implicitly on some model of the world"), as well as the discussion in

«...» of how, were I writing this today, I would use the terminology of 'registration' instead of that of models. (See also the immediately preceding annotation.)

A5 :9/-1:10/0 As noted in «...», for the purpose of this public talk I presented radically simplified views of computing, in order to make the overall claims accessible. Moreover, the "analysis" in this paragraph is not only simplisitic (e.g., in equating computational representations with *descriptions*), but conveys a far more ringing endorsement of the "formal symbol manipulation" construal of computing than I believed even at the time.

Times have also changed. It is no longer considered necessary for programs to represent the structure of the task domains in which they work—especially to represent it explicitly, in a set of language-like formulae or expressions. A great deal of "situated artificial intelligence,"[2] the use of network "models" in dynamic-systems based software, etc., the development of machine learning based on "training" a large set of real-valued nodes and links, etc., which has taken place over the twenty-five years since this paper was written, can be understood as various kinds of attempt exactly to avoid such explicit task domain representation. However: (i) it remains overwhelmingly likely that any software system designed and built to control a major military system of the sort being discussed would still be built on top of an explicit model—if for no other reason than that this design strategy allows the model to be updated, if and as appropriate, when the systems involved change (e.g., the nature and number of missiles, sensors, etc.), without having to build the entire code base over again; and (ii) even machine learning networks and connectionist systems and the like rely on models in the relatively weak sense being employed here[3] (some even develop their own)—it is just that the *representation* of the model in the system may be less explicit that was taken for granted decades ago.

A6 :10/-1/-2:-1 Re "this is not the place for metaphysics": cf. of course O3, AOS, and the discussion of "the ontological wall" in ch. 1 at ....

A7 :11/1/4 The partiality and partial (dis)connection of thinking computation is a major theme of O3; cf. also the forthcoming AOS.

A8 :11/-1/3:4 Though too simplistically formulated, the position underlying this claim is one I would likely endorse even if it were more carefully

[2] «Ref MITACS article.»

[3] Cf. annotation A4, above.

framed. Even to act on a model *as such*, for example, is to act on something whose drenched actuality transcends our registration of it, I believe—even if the model is an abstract mathematical one.

The real difficulty with the statement is its seeming implication that are phenomena that stand in contrast to action—e.g., that reasoning, or inference, or bare computation,[3.5] may somehow "escape" from this property of having consequences beyond those captured in the ways they are registered. This suggestion is one with which I would profoundly disagree (and would have disagreed, even when the paper was written).

A9 ·12/1 The sorts of cognitive resource described in this paragraph are exactly the constituents of reflection that the papers in Part B are an attempt to investigate and explain.

A10 ·14/2/4:8 This claim that mathematics is relatively unproblematic, as regards modeling, is an example of the sort of simplification mentioned in «…» (cf. also annotation «…») that were made for purposes of this being a public talk. Modeling in mathematics is in fact a hugely complex topic—even the merits and demerits of such standard forms of modeling as are used to identify the integers with sets of sets of that cardinality (so that the number two is identified with the set of all sets that have two members, etc.) and the use of Cauchy sequences and Dedekind cuts to model the real numbers are topics that could—and have—occupied entire books.

It might be thought that, even if they are complex, mathematical models of mathematical entities would escape from the general challenges to models being raised in this paper, but I do not believe that that is true, either.

A11 ·:·:· Cf. Eugene Wigner's "The unreasonable effectiveness of mathematics in the natural sciences," *Communications in Pure and Applied Mathematics,* Vol. 13, No. 1 (February 1960). New York: John Wiley & Sons, Inc.

A12 ·14/-1/1:·15:0/4 This question, about the nature and possibility of a semantical analysis of models treating the model-world relationship, was a topic of intense discussion in the early years at CSLI (mid-to-late 1980s), especially between myself and Jon Barwise. Barwise[4] was in principle

[3.5] E.g., the next sentences reference to "more abstract processes of modeling or conceptualization."

[4] Jon was commonly referred to as "Barwise," in part because CSLI was replete with Jon's and John's (Barwise, Perry, Etchemendy, Seely Brown, and others).

interested in studying the model-world relation, and was a strong critic of the unbridled use of models in mathematics itself.[5] Nevertheless, as a committed mathematician, he was unswervingly drawn to use mathematical structures for all semantical analyses, whereas in that period I myself was growing increasingly unhappy with the practice—especially for analyses of the sort being discussed in this paper. Among other reasons, I was troubled by the fact that the identity conditions in mathematics are so much sharper and more definite than anything I felt to be true in the real worlds I wanted to understand (cf. 03).

The issues between Barwise and I went deep, ultimately driving an irrevocable wedge in what up until that point had been a strong sense of intellectual partnership—though we remained friends to the end.

A13 ·16/0/1:2 In describing a program as consisting of "a set of instructions *and* representations,"[6] I was embracing an informal mixture of the ingredientia and specificational views of programs described in ch. 2.

A14 ·16/1/-5:-3 Needless to say, the sentence "Make one milk delivery at each store" is in imperative, not indicative, mood. This is another place[7] where, in deference to this being a public talk, the argument is simplistically phrased. More technically, one might say (it often is said) that a program must specify an *effective procedure* for bringing about a result, whereas a specification must identify the result that is to be produced, but is not subject to the constraint of showing how it can be effectively achieved. Just what 'effective' means, however, is far from clear; the sense is normally conveyed through examples. There is no doubt that programmers develop a keen appreciation of what is and what is not effective; but formulating a theory of computational efficacy is tantamount to develoing a theory of computation tout court—something that, contrary to what is ubiquitously assumed in theoretical computer science, I do not believe we yet possess. See ch. 1, and AOS.

A15 ·17/3/1:3 The sentence is still in imperative mood; cf. annotation A14, above.

A16 ·17/3/3:5 Presumably what you had in mind were some multplicative combinations of 2, 29, 149, and 617.

[5] Cf. annotation A10, above; and «…».

[6] Emphasis added.

[7] Cf. also annotations A… and A…, above.

A17 ·18/3/7 This sentence was so simply stated that it ended up confusing. What is "sloppy and unclear" is that the requirement failed to exclude negative answers, but it is not obvious—in fact it is not clear that there is a fact of the matter as to—whether that is a problem in the model *per se*, or a problem with the specification stated in terms of that model. Either option is available: one could restrict the model to the natural numbers (all positive), or leave the model as the integers (positive and negative) and restate the requirement along the following lines:

> Given number C, produce *positive* numbers A and B such that A≠1 and B≠1 and A×B=C.

The problem is that the requirement, as stated, leaves the model implicit. A fully formal proof would (or anyway should) explicitly identify the intended model—but it is easy to imagine something taken to be a proof in which that decision remained implicit. For all the reasons explicated in the text, what it is to be a "proof" is as prone to error, presupposition, and false assumption as any other issue. Formality, proof, mathematics, etc., are not watertight notions; at the bottom, there are inevitably sieves.

Cf. also annotation A..., below.

A18 ·19/2/-2:-1 That is: there is no reason to believe that the problem of the program's responding inappropriately to lunar reflections would have emerged *in the course of developing the proof*. They would still have emerged, as in fact they did, on the fateful night.

A19 ·20/1/3 Though the term 'Celsius' was formally adopted in 1948, long before the paper was written, 'Centigrade' was still in common use at the time—far more then than at present.

A20 ·20/-1:21/0 The discussion in this paragraph shifts in various ways—focusing first on constraints for a supplier (which requires ruling out strange contraptions), then on what would be required in order for the specifications to ensure that the refrigerator works *correctly*, which brings up the fact that customers, as well as suppliers, are normatively bound by (or in) a background set of tacit assumptions about appropriate use. The differences could be sorted out, but the overall thrust should be clear enough.

A21 ·21/1/8:13 «Ref»

A22 ·22/0/7 «Ref; probably either in *Bringing Design to Software* or in *Understanding Computers and Cognition : A New Foundation for Design*.»

A23 ·25/1/7:9 As is evident from “The Foundations of Computing” (ch. 1) and AOS, not only do I not believe that machines are barred from shouldering responsibility; I am relatively sure that we will, in due course, construct responsible artifacts as a matter of course—or rather, more carefully, that we will construct devices, other than through the traditional nine-month method, that are capable of taking up an intentional poistion within our shared, normatively drenched, society in such a way as to come to carry various kinds of repsonsibilty for their actions. As indicated in the next paragraph (·25/-1:·26/0), however: that would not matter, vis-a-vis correctness. Such systems would no more be “correct” than we are—and should no more be asked to decide the fate of Europe in six seconds that should any member of the species *homo sapiens*.

A24 ·26/0/3 “1000 years” was rhetorical flourish; I did not then, and do not now, think it will take a millennium to develop synthetic devices capable of shouldering genuine responsibility.

More seriously, two overlapping reasons suggest that the emergence of genuine responsibility in artefacts (leaving aside, for the sorts of reasons explored in ch. 1, the question of whether such systems will be *computers*) will be a gradual and incremental process, rather than anything remotely like a singular event. First, responsibility is a vastly complex phenomenon—as many-splendored as any dimension of the human condition. Not only is it far from binary; it would be harrowingly reductive even to describe it is a “matter of degree.” By the same token, and in spite of the ubiquity of the phrase, there is no sense in which humans can be remotely claimed to assume “full responsibility”—whatever such a notion could be made out to mean. Against such a background, it is overwhelmingly likely that the responsibilities shouldered by synthetic devices will emerge gradually, in a piece-meal, incremental, and not necessarily even particularly explicable fashion. Second, and considerably complicating the first point, it is far more likely that the distinction between people and “machines” (i.e., synthesized systems) will blur, possibly even to oblivion, long before anything like adult levels of responsibility are carried by purely synthetic processes or devices. We will all be cyborgs long before substantial responsibility weighs on pure alterity. Note, took, that this second issue will wreak havoc with human notions of censure, punishment, threat, stakes, etc., currently felt to be constitutive of at least the practice, if not the character, of responsibility.

A25 ·26/0/7:8 It would have been better if the paper had said "models must abstract, in order to be useful," rather than that "models must *be* abstract." The point has to do with the inherent violence done to the unutterable richness of the ineffable world by any act of conceptualisation or expression; cf. "The Nonconceptual World," ch. … in *Volume II*; and 03.