Paradigms of Interpretation and Modeling

Peter Wegner

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-91-09
January 1991

Paradigms of Interpretation and Modeling

Peter Wegner, Brown University, January 1991

Table of Contents

Abstract:

In reviewing Umberto Eco's *The Limits of Interpretation* (Indiana University Press, 1990), we explore parallels with interpretation in science and engineering and then focus on modeling and interpretation in computer science. Interpretation in both the humanities and the sciences requires a *model* employed by an *agent* to define *meaning* for a *modeled world*. Realists believe in the reality of the modeled world, formalists in the reality of the model, and idealists in a reality independent of the modeled world and the model. The tension between realists and formalists takes a remarkably similar form in every discipline, focusing attention on epistemological issues fundamental to all branches of knowledge. We examine the literary interpretation of texts, analytic models of science, and synthetic models of engineering. Computer science spans an exceptionally broad spectrum of modeling paradigms, since it employs engineering, empirical, and mathematical techniques. In exploring modeling and interpretation in computer science we focus on models of computation, paradigms of computation, types versus values, and computational versus software complexity.

Paradigms of Interpretation and Modeling

Peter Wegner, Brown University, January 1991

Interpretation in both the humanities and the sciences requires a *model* employed by an *agent* to define *meaning* for a *modeled world*. The modeled world may be physical, conceptual, or fictional, while the agent may be a human interpreter of texts or a mechanical interpreter of programs. Meaning may be objectively inherent in the modeled world or subjectively determined by the agent or model. Texts have an inherent objective interpretation, while their subjective interpretation reflects the agenda of the reader. In reviewing Umberto Eco's *The Limits of Interpretation* (Indiana University Press, 1990), we explore parallels with interpretation in science and engineering and then focus on modeling and interpretation in computer science.

1. Paradigms of Textual Interpretation

Eco's new book of essays on the interpretation of literary texts advocates a balance between the rights of texts and the rights of their interpreters. In conceding that deconstructionists may have overstressed the rights of interpreters he retreats from the position of *open interpretation* of his earlier book *Opera Aperta* (English translation, Harvard University Press, 1989). However, Eco's title has two readings, depending on whether "limits of interpretation" apply to the reader or the text. Does Eco wish to limit the power of interpreters or the power of texts to constrain interpretation? While Eco's pun seems inadvertent, it captures the tension between open and limited interpretation that permeates his book.

The opening essay traces the influence of medieval scriptural exegesis and Renaissance mystical interpretation on modern open interpretation. The second shows that *hermetic drift* by chains of analogies quickly leads to unlimited interpretive freedom. The third essay legitimates variant readings such as semantic versus critical reading and interpretation versus use. The fourth suggests possible-world semantics for interpreting incomplete "small" worlds of fiction. The remaining essays explore a variety of topics ranging from the interpretation of drama to the ontological status of truth. We single out two topics, abduction and encyclopaedia models, because of their relevance to interpretation and modeling in other disciplines.

C. S. Peirce introduced deduction, induction, and abduction as three fundamental modes of reasoning. Deduction is reasoning from assumptions by rules of inference, induction is reasoning by generalization from experience, and abduction is discovery of an explanation for a given set of facts. Abduction, the most novel and powerful of these, involves guessing (or being told) the hidden (black-box) structure that explains or causes behavior. Eco's analysis of Borges' detective story "Six Problems for Don Isidro" shows that knowledge of the solution yields an entirely new reading by revealing the author's abductive model. The knowledge that a text is satirical can reverse its interpretation. In postulating causes to explain effects and providing explanations of abstract behavior, abduction inverts the process of abstraction.

Deconstructionists have hesitated to take on Joyce because his free use of language makes superposition of the additional freedom of open interpretation problematic. Eco contrasts the *dictionary model* of interpretation by local definitions with the *encyclopaedia model* of interpretation by global interconnection. He suggests that Joyce's stream-of-consciousness style in *Finnegans Wake* becomes less arbitrary when the encyclopaedia model is used to determine meaning, since local associations become anchored by multiple interconnections in a larger network. However, he illustrates by examples from *Finnegans Wake* that erroneous interpretation can occur even when ambiguity is narrowed by use of the encyclopaedia model.

The choice between local (dictionary) and global (encyclopaedia) models also arises in physics and other disciplines. Dictionary models are simple and practical, while encyclopaedia

models (action at a distance) are potentially more descriptive but open up so many possibilities that they become intractable. Roger Penrose (*The Emperor's New Mind*, Oxford, 1990) shows that encyclopaedia models of the universe (such as Penrose tiling) require noncomputable physical laws. Penrose also speculates that the human brain, whose multifaceted interconnections suggest the encyclopaedia paradigm, may be subject to noncomputable laws. The encyclopaedia paradigm offers the potential for deeper kinds of meaning, but modeling such meanings may be beyond the combinatorial capacity of humans or computers.

Although Eco traces the pedigree of open interpretation to medieval and Renaissance sources, its emergence in the 20th century can be traced to Pound and Eliot who argued in the 1920s that poetry (for example, Eliot's *The Waste Land*) becomes autonomous once the author completes it. These notions of semantic autonomy influenced Heidegger and Jungian psychologists who created a receptive environment for post-World-War-II open interpretation.

Deconstructionists dethrone traditional basic models of nature and the individual by giving priority to culture over nature and to society over the individual. (This corresponds to the inversion of models over modeled worlds in physics and of types over values in computer science.) Because of this inversion of base and superstructure, R. Harland calls his book on deconstruction *Superstructuralism* (Methuen, 1987). He suggests that superstructuralism is antithetical to the ethic of individualism and the truths of science. It seems paradoxical that freedom of interpretation has spawned theories that devalue the freedom of individuals.

E. D. Hirsch's *Validity of Interpretation* (Yale University Press, 1967) takes issue with open interpretation, arguing that texts have a unique meaning determined by the intention of the author. Hirsch distinguishes between *meaning* and *significance* (Frege's *Sinn* and *Bedeutung*), demonstrating that texts with different meanings may have the same significance ("Scott" and "the author of *Waverley*"), and that a text with given meaning may have many significations.. He argues for the primacy of meaning over significance, while deconstructionists value significance and deny the meaning of meaning. The line between the dominance of significance over meaning and the dominance of opinion over fact is a thin one indeed.

Openness provides opportunities that can be intellectually exciting, but freedom must be balanced by responsibility. Openness in Eco's hands creates intellectual excitement, but has been used by some scholars in irresponsible ways. Its undisciplined freedom can lead to indifferent scholarship and to socially harmful philosophies and political movements.

2. Realism Versus Formalism

The humanities deal with the interpretation of literary texts, the sciences with the interpretation of sense impressions about a presumed chemical and physical world, mathematics with the interpretation of formulae, and computer science with the interpretation of programs. There are remarkable interdisciplinary parallels among paradigms of interpretation.

We introduce three views of the nature of reality which we call realism, formalism, and idealism. *Realism* affirms the reality of the modeled world and promotes the rights of what is interpreted, while *formalism* ascribes reality to the model and promotes the rights of interpreters. The plumber who fixes pipes is a realist, while the theoretical physicist tends to be an formalist. Einstein, whose theory of relativity was formalist, became a realist in later life; he is on record as saying that if he lived his life over again he would be a plumber.

Idealism denies reality of both the modeled world and the model, ascribing reality to metaphysical "ideal" objects. Plato denies the reality of directly perceived objects like tables, believing in the reality of *universals* (ideals) like tablehood, while bishop Berkeley believes that reality is in the mind of God. There is a strong tradition of idealist textual interpretation, but idealism is not our primary concern, since we focus on modeling and modeled worlds. However, many absorbing questions about the nature of reality first asked by idealists are also of central concern

to the epistemology of modeling.

Berkeley's question whether objects continue to exist when there is no one to observe them has a realist answer (yes) and an formalist answer (no). Both Berkeley's and Plato's answer to this question is the same as that of the realists because they believe in an objective reality independent of the observer. Berkeley's answer is yes because objects continue to exist in the mind of God, while Plato questions the existence of the perceived objects, but believes in the independent existence of ideals. In this case idealists are grouped with realists because they both believe in a reality independent of the model. However, in distinguishing those who believe in the reality of the modeled world from those who do not, idealists are grouped with formalists.

Plato believes that we are prisoners of our sense impressions, cave dwellers whose only perception of reality is the shadows on the walls of their cave. He shares with realists a belief in a reality beyond the shadows. But realists find that reality in the modeled world, while Plato denies the reality of both shadows and the modeled world, finding reality in ideal objects. Realist (non-Platonic) cave dwellers believe reflections to be approximations that can be used to infer properties of the real world. The idea that models are an approximation to the modeled world is a powerful metaphor for both physical and computational models.

The reality of realists depends on the nature of the modeled world, while that of formalists depends on the nature of the model. The reality of physicists is physical, while that of mathematicians is mathematical. Newton was a realist in believing that physical laws describe a physical world, while quantum theory ascribes greater reality to observations than to what is observed. Mathematical model theory is realist in that its denotations have independent existence, while constructive mathematics is formalist in denying reality to non constructible entities. Denotational semantics in computer science is rooted in realism (denotations are real), while constructive type theory derives from formalism (types have greater reality than their instances).

Quantum theorists believe that our inability to measure the precise position and momentum of particles is a feature of reality, while Einstein believed that the "real" world is deterministic (God does not play dice with the universe). If we view the position and momentum of particles as Platonic shadows, then quantum theorists are formalists, while those who believe that non-deterministic observations are approximations to a deterministic reality are realists.

Though physical laws constrain the set of possible worlds, they do not necessarily determine a unique reality. Realists go beyond physical laws in presupposing a unique reality even when it cannot be legitimately inferred from physical laws, while formalists accept the abstractions determined by physical laws as reality. Realists are pragmatists who sacrifice certainty and truth in order to create a world that accords with common sense, while formalists are seekers after pure truth. (Note that the term "realist" in everyday conversation effectively means pragmatist.)

Idealist models ignore irrelevant attributes of the modeled world and are therefore weaker than those of realists, who may add pragmatic assumptions not provable from a model. Deconstructionists have weaker models of texts than their realist opponents, believing that the inherent meaning of texts is unknowable. Constructive mathematics has weaker proof systems than traditional mathematics, excluding entities whose existence cannot be proved constructively. (Note that the physical exclusion of entities whose existence cannot be shown by observation parallels the mathematical exclusion of entities whose existence cannot be proved by construction.)

Because provability conflicts with strong models, a choice between the useful and the provably true becomes necessary. Idealism chooses pure truth over pragmatism, while realism chooses the useful over the true. Eco is obsessed with the dangers of seeking pure truth in both *The Name of the Rose* and *Foucault's Pendulum*, portraying the corrosive degeneration of "true believers" into corrupt fanatics. Idealism, by focusing on the provably true at the expense of the pragmatically true, provides moral support for true believers and fanatic fundamentalists.

The modeling paradigms of realism and formalism specialize nicely to Eco's domain of textual interpretation. The belief that texts have inherent meaning is a form of realism, while open interpretation is a form of formalism. Eco stakes out a middle ground, admitting limits that constrain but do not stifle freedom, but leaving open the question of what precisely the limits are.

The conflict between realists and formalists takes a remarkably similar form in every discipline. Recognizing this similarity focuses attention on epistemological issues fundamental to all branches of knowledge. The dichotomy between realist and formalist modeling paradigms is among the deepest epistemological questions both within and among disciplines. Eco's intuitive appreciation of the universality of interpretation augments the power of his writing. He liberally uses interdisciplinary analogies from logic and computing, for example in his discussion of possible worlds and abduction.

Interpretation and modeling focus on powerful ideas in science and the humanities. An undergradute course on this topic could provide a *culturally neutral* introduction to "great ideas" complementary to current courses on "great books". Such a course could range over interpretation and modeling in literature, religion, law, anthropology, economics, biology, physics, mathematics, and computing.

3. Analysis Versus Synthesis

How can we classify modeling techniques? We distinguish between universal criteria applicable to all disciplines, generic criteria applicable to broad classes of disciplines, and specialized modeling techniques in particular disciplines.

universal criteria: objective (realist) versus subjective (formalist) models
generic criteria: scientific (analytic) versus engineering (synthetic) models
specialized criteria: within specific disciplines

Universal criteria are the province of philosophers like Plato and semioticians like Eco. Scholars in particular disciplines generally have a non philosophical mindset that ignores interdisciplinary features of modeling. However, interdisciplinary thinking can help in understanding particular models (for example quantum mechanics and relativity theory). In fact, interdisciplinary concern with the nature of modeling often distinguishes deep scientific results that contribute to paradigm shifts from specialized results in a particular paradigm.

Although science and engineering borrow heavily from each other, they have different goals and consequently different modeling techniques. Scientists describe and develop theories about natural phenomena, while engineers construct man-made (artificial) products. Scientists analyze the real world, while engineers synthesize models of artificial (possible and sometimes impossible) worlds. Scientists are readers whose unique text is the real world, while engineers are authors who synthesize new texts (engineering products). Scientific models tend to be objective, while engineering models tend to be subjective, reflecting particular design agendas.

Engineering design is like writing a book rather than interpreting an existing one. Whereas critics and interpreters devote their energies to analyzing existing texts, authors construct an artificial reality that may be interpreted or used by others. Both authors and engineers must serve a long apprenticeship to master their design medium. However, engineering design differs from artistic creation in being an economically driven rather than artistic activity.

Design is the process of conceptualizing the form and structure of artifacts prior to their construction. Architects, authors, and shipbuilders share a concern with the construction (synthesis) of artifacts. Design is a form of abduction that creates structure for a specification. Whereas literary interpreters *discover* hidden abductive structure, designers and authors *synthesize* abductive structure for black boxes to achieve explicitly specified or implicitly contrived effects.

The distinction between discovery or analysis of a modeled world and synthesis of a new one is fundamental. Both involve modeling and interpretation, but the central role of the interpreter during analysis is usurped by the creator during synthesis. Abduction involves discovery of structure during analysis and creation of structure during synthesis. There is, however, a sense in which synthesis is also discovery (by actualizing a potential possible world).

The relation between creators and interpreters may be clothed in cosmic terms by juxtaposing God the creator to Platonic cave dwellers who interpret created worlds through their reflections. Developers of products and texts generally have less power over their creations than omnipotent creators portrayed in world religions. Creators exercise control over artifacts during their creation, and may possess special powers or rights of interpretation, but lose control once creation has been completed. Note the similarity to Eliot's view that poetry becomes autonomous once the author completes it. Modeling the process of creation, while generally beyond the scope of scientific or textual models, is central to engineering.

4. Modeling and Interpretation in Computer Science

Modeling and interpretation are primary goals of computer science. Modeling applications is the central goal, while interpretation determines the meaning of programs by executing them. Computer science spans an exceptionally broad spectrum of modeling paradigms, since it employs engineering, empirical, and mathematical techniques.

Applied and theoretical models in computer science have differing goals and structure. Application-driven models are the province of software engineering, while models of computation are the province of theory. Software engineering reaches out to other disciplines in solving problems of the real world, while theory is introspectively concerned with properties of algorithms and computational formalisms. Software engineering models assist in the dynamic process of software construction, while theoretical models illuminate static, invariant properties of the process of computation. Like other branches of engineering, software engineering employs synthesis and abduction: the behavior of black boxes given by a specification is implemented by creating an abductive structure that realizes the behavior.

Instead of exploring the differences between applied and theoretical models, we focus on their similarities. Both must deal with the many-to-one relation between the model and the modeled task. The attempt to provide a metric for comparing alternative models yields the notion of software complexity for models of application and of computational complexity for theoretical models. We first examine explore the role of interpretation in models and paradigms of computation and then contrast the notions of software and computational complexity.

4.1. Models of Computation

Alan Turing demonstrated that all models of computation have equivalent modeling power, modeling the class of computable functions. He introduced a particularly simple computer, now called a Turing machine, as a paradigmatic representative of all computers. Digital computers model the computable functions by their primitive instructions, interpreting their machine language. Programming languages model the computable functions by linguistic structures whose conceptual interpreters are equivalent to physical computing engines.

There are many ways of specifying programming language semantics. Operational semantics specifies meaning by an interpreter for program execution, while denotational semantics specifies meaning by a denotation. Denotational semantics specifies *what* is computed, while operational semantics specifies *how* we compute it. "What" is an objective question while "how" is a subjective one. Objective denotations may be computed in many subjective ways.

Denotational semantic models are realist in their goal of mapping programs into unique denotations. However, the question whether denotations are really real or simply another

representation is ever present. Is the reality underlying denotational semantics the mathematical model or is there a nontextual reality beyond the mathematical model? The computational Platonist views denotations as reality itself, while the realist views denotations as possibly incomplete representations of reality.

Operational semantic models are more than overspecified realizations of denotational models. Their detailed structure aids both human and mechanical interpreters. The comparison of operational realizations of a denotational specification is inherently interesting. Plato was wrong to scorn imperfect real-world objects in favor of abstract ideals, since concrete objects have a richer meaning than ideal prototypes.

The operational semantics of programming languages may be defined by interpreters that specify the step-by-step execution of instructions. Interpreted programs are treated as data, their semantics being specified by transformations from an initial to a final data configuration. Program interpreters have simple rules for executing instructions and selecting the next one. The earliest formally specified interpreters were those for Turing machines and finite automata. The programming language Lisp, designed to illustrate fundamental principles of interpretation, has served as a canonical model for operational semantics.

Interpreters for most programming languages are more complex than for Lisp but have essentially the same structure. Precise models of the process of program interpretation provide insight into models of interpretation in other domains of discourse. Computer science is moving beyond the interpretation of linear texts to the interpretation of concurrently executing modules, and to visual programming languages whose interpretation paradigms are more complex than linear paradigms of textual interpretation.

Higher-level languages are in practice interpreted by first translating programs into a low-level machine language and then interpreting the machine-language program. Translation programs, called compilers, have a characteristic structure consisting of syntactic analysis to recognize linguistic constructs followed by translation and possibly optimization. The idea of translating text from one representation into another is quite general, providing another example of precise computational models that relate to fuzzy general notions of interpretation.

The relation between compilers and interpreters can be described in terms of the notions of syntax and semantics. Compilers and interpreters have a common syntactic recognition phase followed by different semantic actions once a language construct is recognized. Compilers statically transform constructs of the source language into behaviorally equivalent target language constructs, while interpreters dynamically execute language constructs. Translation may be viewed as a particular form of interpretation, where the meaning of a program is the target language text rather than the value computed by applying a program to its data.

Syntactic models may be classified into *generative models* that describe the generation of text by speakers or writers and *recognition models* that describe the recognition of text by listeners or readers. Language generation and recognition mechanisms were studied by Chomsky and Miller in the 1960s for natural languages and have been refined for specifying programming languages and their compilers. The use of grammars for the parsing of programming languages is viewed as one of the primary successful applications of theory to practice. Precise models of program generation and recognition provide a basis for a syntactic, though not a semantic, theory of how writers and readers generate and recognize natural language texts.

4.2. Paradigms of Computation

Paradigms partition the design space of models of computation into classes with shared properties, organizing them into manageable categories. Models in the same paradigm share static linguistic structure and dynamic execution structure. The Von Neumann paradigm determines a class of computers with similar language and computation structure, while Turing machines

determine a simpler and purer form of the Von Neumann paradigm. The block-structure paradigm determines a class of programming languages with shared structural properties (nested block structure). The thousands of existing programming languages may be classified into a small number of paradigms with characteristic program structure and execution-time behavior.

The procedure-oriented, object-oriented, functional programming, and logic programming paradigms determine four very different principles for organizing computation: nested procedures, autonomous objects, mathematical functions, and logical relations. These paradigms are equivalent in computational power but make us think about and model computation in different ways. Paradigms of computation are subjective perspectives of the computable functions. The choice among models of computable functions is governed by the degree to which they can be economically effective in software development.

If a task is doable, there are generally an infinite number of ways of doing it. There are a large number of ways of designing a building, a software product, or an algorithm. The differences among sorting algorithms (bubble sort, mergesort, quicksort) are so substantial that it is difficult to demonstrate their equivalence. The number of different programs for performing a given task is generally infinite and the question whether two programs are equivalent is undecidable. Paradigms and models of computation are concerned with a specific particularly hard task: the task of modeling all computable functions.

The object-oriented paradigm has spawned dozens of object-oriented languages in the last decade. It organizes software structures into collections of autonomous objects that reflect the organizational structure of modeled worlds. There is evidence that object-oriented models are often natural and effective representations of reality. But the object-oriented paradigm restricts the design space, excluding procedure-oriented, functional, and logic programming solutions to a task. It is a heuristic technique justified by its high rate of success that may sometimes yield suboptimal models. A multiparadigm modeling technique that allows tasks to be programmed in the paradigm most suited to their inherent structure may be the most effective.

4.3. Types Versus Values

Types are a mechanism for modeling classes of values with shared behavior. Semantic models of language or computation may have values or types as basic modeling primitives, depending on whether objects of the modeled world or elements of the model are viewed as primary. Are types a redundant auxiliary construct to aid the human intellect, like a construction in a geometric proof, or the primary "reality" that determines the meaning of values?

Realists view values as primary and types as redundant though convenient constructs for describing properties of classes of values. The type "integer", which captures common properties of the class of integers, is a useful but redundant abstraction, since the behavior of integers can be defined for each individual integer. Traditional programmers view types as auxiliary entities useful in clarifying meaning but not necessary for defining it.

Idealists view types as primary and values as existing only when constructible from a type. Once a type system for describing a domain of discourse is in place, it takes on a life of its own and may be used as the defining model for behavior. A description of a domain in terms of its types is cleaner than a description in terms of values. It is tempting to adopt the type system as a definition of behavior, so that behavior not derivable from the properties of types becomes inadmissible. This shifts attention from values in the modeled world to the model that describes behavior, just as for formalist models in other disciplines. Types provide an interpretation for values, just as readers provide an interpretation for text. Types are an interpretive filter that colors and defines all meaning in the domain of the model.

Constructive type theory views types as the primary modeling primitives, and values as existing only when constructible from a type. The idea of truth is replaced by the idea of

evidence: a proof of a proposition is evidence for its truth or falsity. Values require proofs that provide evidence for their existence. The set of all proofs of a proposition corresponds to the set of all values of a type. Constructive type theory is used primarily for type inference (inferring the type of expressions in a program). It does not deal directly with the computation of values by programs, but may be useful in proving that programs terminate. Its programming-language models illustrate both the power and the limitations of elevating the notion of types over the notion of values. We gain insight into the properties of types, but may develop models that are too high-level to be relevant to the real problems of computing.

The view of types as finite approximations of infinite values is formalist in giving types priority over values. Values are an unattainable ideal, while types are the model by which values are approximated. In the pure version of this model, computer representations of numbers are finite approximations of infinite ideals and are therefore types. Object-oriented inheritance is a nice real-world example of a type approximation mechanism. Subclasses determine progressively better approximations to behavior by adding new attributes (methods) or refining existing ones. However, in object-oriented programming this ability to refine behavior approximations at the level of classes is supplemented by traditional computation with values.

Types provide a paradigmatic framework for computational modeling, while computational models in turn provide a paradigmatic framework for all models because of their general-purpose ability to model any application domain. Object-oriented types, called classes, are a particular paradigm for types, mathematically rooted in algebra and conceptually supported by principles of scientific method. Algebras with one or more sorts (the state set) and a collection of operations have the mathematical structure of computational classes. Conceptually, the behavior of objects is naturally modeled by abstractions that specify operations (methods) applicable to an object while hiding its state. Inheritance facilities the enhancement of behavior, providing for behavior refinement and progressive approximation.

Ascribing greater reality to types than to values conforms to influential philosophical traditions. It reflects Kant's distinction between empirical concepts which are synthetic, and "a priori" categories of the mind, which serve to constrain the form empirical phenomena must take. Kant asserts that the world must conform to the structure of the mind, as opposed to the mind conforming to the structure of the world. This is an formalist position, with Kantian categories playing the role of types.

The shift from values to types reflects a process of maturing from direct to higher-level models that occurs in many disciplines. However, mature models may cause the modeler to lose touch with reality and operate in an ideal world whose relevance to the real world must constantly be reestablished. This is a perennial dilemma for the "pure" scientist or mathematician. Models that abstract away irrelevant attributes cannot completely replace the modeled world because attributes deemed irrelevant in one context may be relevant in another.

Early models in any discipline are generally realist. In science the paradigm of observing regularities, developing predictve theories, and verifying predictions is realist. But once theories and models are developed, they themselves become objects of study, and the phenomena being modeled may become less central. Subjective (formalist) models replace concerns with the modeled world by concerns with modeling abstractions. Type theory illustrates the replacement of the modeled world of values with properties of models of types.

However, practical modeling systems should support both the realist and formalist paradigms. They should support type specification and approximations as well as computation on values. Although finite representations of values are technically approximations to infinite ideals, they are in practice sufficiently accurate to be treated as reality itself. Plato was wrong to conclude that we have no power over reality because we cannot perceive or completely represent it. Realists may be formally wrong in treating approximate numbers as real values, but their

pragmatism is justified by results.

4.4. Computational Complexity Versus Software Complexity

The notions of complexity in software engineering and the theory of computation are so different that each community regards use of the term by the other as unfortunate. But both measure the comparative difficulty of alternative ways of performing a given task. Computational complexity provides a precise measure of resource costs (usually running time) as a function of problem size, while software complexity aims to measure the economic cost and psychological difficulty of constructing and maintaining software. The elegance and quantitative precision of computational complexity contrasts sharply with the imprecise and often ineffective heuristics for controlling software complexity. But the common goals of computational and software complexity justify the use of a common term.

Complexity theory includes *possibility results* that provide upper bounds on the running time of a task by exhibiting an algorithm for the task. It also includes *impossibility results* that provide lower bounds on all possible algorithms for a given task. Lower bounds are much harder to establish than upper bounds since they require proving something for all possible algorithms, and the class of all possible algorithms for a task is very hard to characterize.

The class of all problems computable within a given time or resource bound is called a *complexity class*. Complexity classes are semantically neutral, imposing a very coarse equivalence relation that does not distinguish between algorithms with very different behavior but comparable running times. Moreover, algorithms with the same behavior may be in different complexity classes (sorting algorithms with a time bound of $n\uparrow 2$ are in a different class from those with a time bound of N logN). The classification of algorithms into complexity classes is a primary goal of complexity theory. Two complexity classes are said to be equivalent if any tasks computable in one of the classes is also computable in the other. The problem of determining whether or not complexity classes are equivalent has turned out to be surprisingly hard and deep.

The complexity class P is the class of all tasks (problems) that can be performed in polynomial time on a standard computational model, while the class NP is the class of all tasks that can be performed in nondeterministic polynomial time. Note that the classes P and NP are very coarse equivalence classes (algorithms computable in time N and $N\uparrow 2$ are both in P). Playing perfect chess is an example of a task that is certainly in the class NP and does not appear to be in the class P. Perfect chess can be played in linear nondeterministic time (but exponential space and processors) by simply generating the tree of all possible moves, but appears to require exponential time when computations are restricted to be deterministic.

The question of whether chess is in the class P is still open, as is more general question of whether NP is larger than P (is nondeterministic computation more powerful than deterministic computation?). The problem of whether or not P = NP is considered to be a fundamental problem of complexity theory and a paradigmatic example of a deep problem in computer science. It appears intuitively unlikely that nondeterministically simple problems like playing perfect chess or proving theorems can be reduced to deterministically simple problems, but the power and fascination of complexity theory are due in part to its ability to prove nonintuitive results.

Computational complexity is among the most intellectually challenging areas of theoretical computer science. But as the ratio of software development to hardware costs increases, classical computational complexity becomes less important economically than software development complexity. Moreover, lower computational complexity may result in higher conceptual complexity, since extra efficiency may require conceptual subtlety.

Problems of software complexity are beyond the current scope of mainstream theoretical computer science, but are central to software engineering. The central problem of software complexity is the construction of complex systems from simple components and composition rules.

This paradigm, which derives originally from Euclidean geometry, was generalized to formulae and rules of inference of formal systems, and was adapted to the static generation of programs from primitive linguistic constituents and dynamic application of rules of semantic composition to define their effect during execution. By constructing programs from primitive expressions and statements and specifying their semantics by rules for their sequential execution, we can specify complex computational effects in terms of simple primitives.

Abstractions with interfaces that encapsulate hidden information increase the complexity of primitives and composition rules. Software components are more powerful in their primitive behavior, and the composition of software components goes beyond imperative sequential composition of transformations to declarative composition of behaviors. Component composition specifies the declarative interaction of loosely coupled, autonomous software components rather than the imperative execution of sequences of statements.

The declarative extension of the composition paradigm to data abstractions and other software components is the basis of component-based software technology. We do not have a theory of complexity for component-based software technology to match complexity theory for algorithms. But methodologies for software design that facilitate the construction and management of component-based software structures are becoming increasingly effective.

Software design is the task of decomposing a requirement specification into a collection of components by which they may be realized. The designer is given a design language whose primitives and composition rules constrain the inverse process of decomposing requirements into software components. The language for specifying complexity by composition must be chosen to facilitate the inverse process of decomposition. Decomposition is an abductive process, creating internal structure for the black box specified by the requirements. One of the primary goals of software engineering is to transform design by decomposition from its current status as an abductive art into a systematic engineering technique.

Acknowledgements: This paper has benefited from discussions with Hassan Ait Kaci, Joseph Dan, Ernie Frerichs, David Hirsch, Franco Preparata, Peter Richardson, and Billy Wooten.