Tossing Algebraic Flowers down the Great Divide

Joseph Goguen
Dept. Computer Science & Engineering, University of California at San Diego

1 Introduction

Computer science today is extraordinarily successful; chips are reproducing and evolving far faster than humans, and millions of humans are exchanging email, visiting websites, and discussing HTML, Java, and high speed modems in cybercafes across the world. Computers are the only significant commodity to ever get progressively cheaper as they get better, throughout their entire history.

But computer science is in deep crisis, expanding, fragmenting and specializing faster than any other discipline, faster than anyone can understand, let alone predict. Moreover, computer science is increasingly seen as marginal to its applications, and this is particularly true of theoretical computer science.

Information is the life blood of modern society, and much of it is managed and distributed by computer systems; they control cash machines, factories, nuclear reactors, telecommunication networks, and ballistic missiles, as well as arcade games, the family car brakes, and an almost unimaginable variety of databases, e.g. for health records, country music hits, vehicle registrations, stock transactions, and DNA sequences.

Nobody knows where all this is going or what it will mean for people's lives; those in government who are responsible for overseeing technology and its impact are often remarkably ignorant. This is due not only to the unprecedented growth of information technology, but also to the unprecedented nature of its relationship to society: Information only has value insofar as it has *meaning*, i.e., is about something, whether money, braking distances, transaction costs, or genetics. Otherwise, it is just *data*, patterns of bits, strings of characters, etc. Data can only become *information* when people care about it for some reason and are able to interpret it. This means that information technology, and thus computer science, is bound up with the social at a very basic level having to do with the nature of information itself.

Most work in theoretical computer science ignores all this. I would never suggest that theory has no value, but I do suggest that this "great divide" between theory and practice helps explain the ever declining interest shown in theory. The rest of this paper explores this theme in various ways, beginning with a classification of the fragments – we might say cultures – of computing, and then moving to more detailed consideration of the interplay between theory and practice in certain areas.

This paper can also be considered a survey of some attempts to bridge the great divide between theory and practice in computing, mainly using various kinds of algebra, as well as to bridge the "even greater divide" between technical and social aspects of computing, which in turn is but a small part of the huge rift between science and technology on one side, and society on the other (see [6] for more on this).

From another perspective, this paper can be considered a diary from a very personal journey[1], moving from a mathematical view of computing, through a process of questioning why it wasn't working as hoped, to a wider view that tries to integrate the technical and social dimensions of computing. This journey has required a struggle to acquire and apply a range of skills that I could never have imagined would be relevant to computer science. Always I have sought to discover things of beauty – "flowers" – and present them in a way that could benefit all beings, though of course I don't expect that very many people will share my aesthetics or my ethics. I also hope that this piece may help younger researchers to see more of the process and the human context of research; for that reason I have tried to bring out attitudes as well as facts. Although this paper is by no means intended as a general survey, I have still tried to be as fair as possible to everyone; but through the years, I have discovered that sins of omission and misattribution are inevitable, especially in such a very personal path through such a broad landscape, and so I apologize and invite readers to send me corrections and additions that they consider important.

[1] I hope the large number of citations that signpost stages along this journey won't be thought too egomaniacal; the truth is that this essay evolved out of one of those dreadful documents that (many) professors have to write to get a salary increase, so that it was easier to leave all these citations in than to take them out. And who knows, someone might find some of them useful.

1.1 Five Cultures of Computing

For present purposes, we may identify five cultural fragments of computing, each involving a different group of people, characterized by different goals and different activities:

1. **Computer hardware** Hardware engineers have been phenomenally successful. As this is written, an ordinary high-end PC runs at 333 MHz, has 128 Megabytes of RAM, and 8 gigabits of hard disc (which is probably more computing power than all the computers in the world of twenty-five years ago combined), along with peripherals that would once have been astonishing, including FAX, CD quality digital stereo, and real time audio and video over the internet; by the time you read this, there will most likely have been further advances. Twenty-five years ago, the arpanet had only a few hundred users, and for the most part we all knew each other; now the internet has millions of users, including many we would probably not care to know.
2. **Computer software** Software engineers have been less fortunate than hardware engineers in their choice of profession. The fantastic improvements in hardware have fueled escalating expectations for software that are not being met. Huge "legacy" systems, often written in obsolete and/or obscure languages, with little documentation and generations of superimposed "patches," are exceedingly difficult to maintain, but are surprisingly common. And numerous expensive software failures have been reported, at the Federal Aviation Agency, the Internal Revenue Service, the European Space Agency, and more, as discussed in Section 2.1.
3. **Shrinkwrap computing** Consumers are beneficiaries of the great success of hardware as well as victims of the doubtful state of software. There is an immense popular culture of computing, with dozens of magazines on the racks of supermarkets, record stores, and drug stores; many newspapers have weekly features on computing, often a special pullout section; there are also very many popular books, and even TV and radio shows. But shrinkwrap software is typically badly documented, full of bugs, hard to understand, and hard to use. It is also notoriously quickly replaced by a "new and improved" version, or even an entirely new system, often requiring more powerful hardware, and of course providing new bugs.
4. **Sociology of computing** The study of computing by psychologists, sociologists and anthropologists is a fairly new and still relatively small phenomenon. The relevance of such work to user interface design and team management is clear, but I will argue in Section 3 that it also has a much broader relevance to computing.
5. **Theoretical computer science** Unfortunately, the explosive growth of computing practice has had little effect on theoretical computer science, which continues to decline in relative importance within industrial and even academic circles, despite many impressive advances in its own terms.

It seems that these five cultures[2] are rather isolated from each other, having quite distinct conferences, journals, methods, and goals. Of course the boundaries are somewhat vague and overlapping, but it does seem to me that they are moving away from each other at an increasing rate, and that each one itself is becoming increasingly fragmented. As a result, there is a great need for communication and even unification between these cultures as well as within them. My major theses are (1) that computer science has not paid sufficient attention to social issues, and (2) that theoretical computer science could play a key role in a reunification that would yield significant benefits to both society and education.

2 Software Engineering

Beginning with a closer look at the state of software engineering, the subsections below sketch some ways to apply algebra to problems related to software engineering. These include: general system and sheaf theories; abstract data types; specification languages; logical programming and institutions; parameterized programming; order sorted algebra; hidden algebra; specification libraries and reuse; and distributed cooperative engineering. In each case I try to indicate the original motivation, and how I now see this in terms of larger concerns about relations between technology and society.

[2]A wag might want to call them hardware, software, shrinkware, wetware and airware, respectively.

2.1 State of the Software Arts

Large complex software systems fail much more often than seems to be generally recognized. Perhaps the most common case is that a project is simply cancelled before completion. This may be due to time and/or cost overruns or other management difficulties that seem insurmountable; it may be due to politics; it might even be due to purely technical difficulties. One highly visible example is the cancellation by the US Federal Aviation Agency of an $8 billion contract from IBM to build the next generation air traffic control system for the entire country [158]. This is perhaps the largest default in history, but there are many more examples, including cancellation by the US Department of Defense of a $2 billion contract with IBM to provide modern information systems to replace myriads of obsolete, incompatible systems. Other highly publicized failures include IBM software to deliver real time sports data to the media at the 1996 Olympic Games in Atlanta, the $2 billion loss of the European Ariane 5 satellite, and the failure of theUnited Airlines baggage delivery system at Denver International Airport, delaying its opening by one and a half years [30].

What these examples have in common is that they were hard to hide. Anyone who has worked in the software industry has seen numerous examples of projects that were over time, over cost, or failed to meet crucial requirements, and hence were cancelled, curtailed, diverted, replaced, or released anyway, sometimes with dire consequences, and sometimes with a loud declaration of success, even though the system was never used, and may well have been unusable. For obvious reasons, the organizations involved usually try to hide their failures, but experience suggests that half or more of large complex systems fail in some significant way, and that the frightening list in the previous paragraph is just the tip of an enormous iceberg. Much more information about computer system failures can be found in the Risks Forum run by Peter Neumann (see `http://www.csl.sri.com/neumann.html` and [154]).

Experience shows that many failures are due to a mismatch between the social and technical aspects of a supposed solution. It is understandable that software engineering has been biased towards a formal view of information, because computer programs consist of precise instructions that manipulate formally defined structured representations of data, and this is what software engineers are trained to deal with, as opposed to relatively more messy social situations. But we now know that ignoring the situated, social aspect of information can be fatal in designing and building software systems.

2.2 Category and General System Theories

In the late 1960s, I greatly admired the smooth way that category theory captured many important general concepts in mathematics (e.g., see [128, 129]), and I greatly regretted the lack of a similar apparatus for engineering. My first attempt to use categories was in my thesis, which gave axioms for fuzzy set theory (see Section 3.4). Because of this enthusiasm, I wrote several introductions to category theory for computer scientists, beginning in the early 1970s with the "ADJ"[3] report series [107, 108, 109], illustrating basic categorical concepts mainly with examples from automata and formal languages, which were the focus of theoretical computer science at that time. I've been told that many East Europeans of that generation learned both basic category theory and theoretical computer science from these reports. Our original goal was to write one or more comprehensive books, something like what Bourbaki did for mathematics, but ADJ fell apart before we got any further than these reports. Later I wrote "A Categorical Manifesto" [56] to provide for each basic concept of category theory a "doctrine" of how to use it in practice, and still later, [53] developed category theory from scratch while proposing a general theory of unification.

Following clues from the systems engineering, general systems theory (hereafter **GST**) and cybernetics of the late 1960s, I decided that the most general concepts of engineering might be system, behavior, and interconnection, formalized in such a way as to include hierarchical whole/part relationships. Systems were taken to be diagrams in a category, behaviors were given by their limits, and interconnections were given by colimits of diagrams; some very general laws about interconnection and behavior hold in this setting [35, 37, 78]. The most complete exposition is in [62], which has full proofs of all results.

[3]The ADJ group, {Goguen, Thatcher, Wagner, Wright}, was formed during my tenure as Research Fellow in the Mathematical Sciences at IBM Research, Yorktown Heights, initially to study the relationship between category theory and computer science; see [51] for many historical details; for some reason, I wrote all of the initial reports, but in compensation, had very little role in some of the final reports.

One especially nice feature of this approach is that it does not build in any notion of causality, and therefore can capture the sort of mutual interdependence that occurs, for example, in electrical circuits. This contrasts with models like automata in which a causal dependence of the next state on the current state and current input is built into the model. It is also consistent with philosophical ideas like mutual causation and interdependent origination[4], which go beyond naive reductionist causality. But it was (and still is) disappointing to me that so few people felt any need for concepts and theories of such generality; they seem happy to have (more or less) precise ideas about specific systems or small classes of systems, with little concern for what concepts like system, behavior and interconnection might actually mean. Still, this categorical GST has had a significant indirect impact on computer science: its application to the Clear and OBJ module systems influenced some important programming languages, including Ada, ML and C++ (see Section 2.6).

A general theory of objects based on sheaf theory [40] arose from this work, and has been applied (for example) to the semantics of concurrent systems, including concurrent object based languages [62], hardware description languages [166], and semantics for object based concurrent information systems [23]. Sheaves can express the kind of local causality combined with global non-determinism that characterizes many different kinds of model, from partial differential equations to automata. They can capture not only variation over time, but also over space and over space-time [62]; and they can embrace the incompleteness of observation that is characteristic of all real empirical work. This can be helpful at the philosophical level in dispelling the illusion that models fully capture reality (see the discussion in Section 4.1). It can also be useful technically, for example in capturing the way that the behavior of distributed concurrent systems depends only on local interactions. There is now a slow but steady stream of research on sheaf theory in computer science, though there is not as yet a coherent community.

In the early 1970s, I formulated the minimal realization of automata as an adjoint functor [38]; this soon evolved into much more general results about the minimal realization of machines in categories, which gave a neat unification of system theory (in the sense of electrical engineering) with automaton theory [36]. I consider this a major vindication of the categorical approach to systems.

2.3 Abstract Data Types and Algebraic Semantics

The history of programming languages, and to a large extent of software engineering as a whole, can be seen as a succession of ever more powerful abstraction mechanisms. The first stored program computers were programmed in binary, which soon gave way to assembly languages that allowed symbolic codes for operations and addresses. FORTRAN began the spread of "high level" programming languages, though at the time it was strongly opposed by many assembly programmers; important features that developed later include blocks, recursive procedures, flexible types, classes, inheritance, modules, and genericity. Without going into the philosophical problems raised by abstraction (which in view of the discussion of realism in Section 4 may be considerable), it seems clear that the mathematics used to describe programming concepts should in general get more abstract as the programming concepts get more abstract. Nevertheless, there has been great resistance to using even abstract algebra, let alone category theory.

One of the most important features of modern programming is abstract data types (hereafter, **ADT**s), which encapsulate some data within a module, providing access to it only through operations that are associated with the module. This idea seems to have been first suggested by David Parnas [155, 156] as a way to make large programs more manageable, because changes will be confined to the inside of the module, instead of being scattered throughout the code. For example, if dates had been encapsulated as an ADT, the so-called "year two thousand problem" would not exist. Not all increases in abstraction make programming easier; an abstraction must match the way programmers think, or it won't help. This may explain why ADTs have been more successful than higher order functions.

In the early 1970s, there was no precise semantics for ADTs, so it was impossible to verify the correctness

[4]The concept of interdependent origination goes back over 2,500 years to the Buddha; in the Pali language, it is called *paṭicca-samuppāda* [12]. This kind of thinking can also be found in much contemporary AI, e.g., the robotics of Rodney Brooks, which is intended to be fast and cheap, because it doesn't require any central control (cf. the wonderful documentary movie *Fast, Cheap and out of Control*, directed by Errol Morris).

of an implementation for a module, or even to formulate what correctness means. Initial algebra semantics provided the first rigorous formulation of these problems, with solutions that were useful, although they have been improved (see Section 2.10). Initial algebra semantics was born in [41], which (among other things) formulated (Knuthian) attribute semantics as a homomorphism from an initial many sorted syntactic algebra generated by a context free grammar, to a semantic algebra[5]. The step to ADTs was facilitated by my realization that Lawvere's characterization of the natural numbers as an initial algebra [132] could be extended to other data structures [105, 106]. As a young researcher at this time, I was really shocked by the attempts of certain senior colleagues to reconfigure the history of this period to their own advantage; this is why I wrote the paper [51].

What really pleased me was the neat parallel established between Emmy Noether's insight that algebra is the study of sets with structure given by operations, and David Parnas's insight that modules should encapsulate data with operations; more than that, the algebraic approach established an equivalence between abstractness in ADTs and abstractness in algebra[6]; furthermore, equations among operations came into specifications of ADTs the same way as in abstract algebra.

It also seemed splendid that computability properties worked out so well: an algebraic version of the Turing-Church thesis says that an algebra is computable iff it is a reduct of a finitely presented initial algebra with an equationally definable equality; these are also the algebras for which so-called "inductionless induction" proofs are valid [47]. Moreover, an algebra is: semicomputable iff it is a reduct of a finitely presented initial algebra; cosemicomputable iff it is a reduct of a finitely presented final algebra; and computable iff a reduct of a finitely presented algebra that is both initial and final. The reason that the computability notion associated with Scott-style denotational semantics doesn't work for algebras is explained in [46]. This field was pioneered in a series of papers by Jan Bergstra and John Tucker, surveyed in [150], which also explains basic many sorted algebra and abstract machines. Some conjectures from [150] were solved with Meseguer and Moss in [153]. The computational side of ADTs includes term rewriting, which featured in early drafts of [106]. Initial algebra semantics has also been used in linguistics, to explicate to the notion of compositionality [122].

Meseguer and I studied the rules of deduction for many sorted algebra in [94]. This paper surprised the community by showing that the naive generalization of the usual unsorted rules (as previously used in the ADT literature) is unsound. We gave a sound and complete set of rules, and showed that the unsorted rules did work for certain signatures; the difficulties involve implicit universal quantification over empty sorts. Complete rules of deduction for many sorted conditional equational logic are given in [92]. The first rigorous proof of correctness for the inductionless induction proof technique that was originally suggested by David Musser, is given in [47]. The formulation of inductionless induction in [47] is more general than in some later work, which was restricted to just constructors; [47] also pointed out that the essence of inductionless induction is "proof by consistency," and gave the simple but fundamental result that for an equational specification that is canonical (i.e., terminating and Church-Rosser) as a set of rewrite rules, the normal forms of ground terms form an initial algebra. This result justifies term rewriting as an operational semantics for initial algebra semantics.

ADJ later extended initial algebra semantics to continuous algebras (at about the same time as Maurice Nivat) and continuous algebraic theories [110], and then to rational algebraic theories [173]. This inspired Meseguer and me to generalize to an arbitrary category, getting *initial model semantics* (see Section 2.9).

2.4 Specification

In the early seventies, most theoretical research concerned the semantics of programs and programming languages, and the verification of programs. There was little or no work on specification, modules, or

[5] This built on an approach to many sorted algebra developed for my course Information Science 329, Algebraic Foundations of Computer Science, first taught in 1969 at the University of Chicago, including the now familiar use of indexed sets, the word 'signature' with symbol Σ, and its formal definition.

[6] This is because any two initial objects in a category are isomorphic; hence we speak of "the" abstract data type of a specification in exactly the same way that we speak of "the" initial algebra of a variety. Each is determined up to isomorphism, and the fact that each an "abstract algebra" and an "abstract data type" are isomorphism classes of algebras expresses their independence of representation.

verification at these levels. But we now know, and even then many suspected, that this is not where the leverage lies for real applications; in fact, most debugging effort goes into fixing errors in requirements, specifications and designs, and very little into fixing errors in coding (around 5%) [5]. Moreover, the problems that arise for large programs are qualitatively very different from those that arise for small programs. It is *not* just as easy to find specifications and invariants for the flight control software of a real airplane as it is for a sorting algorithm; in fact, finding specifications and invariants is not an important activity in real industrial work. On the contrary, it turns out that finding requirements (i.e., determining what kind of system to build), structuring the system (modular design), understanding what has already been done (reading documentation and talking to others), and organizing the efforts of a large team, are all much more important for a large system development effort. As Tony Hoare said about his research (largely on program correctness), "It has turned out that the world just does not suffer significantly from the kind of problem that our research was originally intended to solve" [120].

I thought that since we knew how to do ADTs as theories, the next step (according to categorical GST) should be to interconnect these theories using colimits; then a description of such an interconnection would be a design for a system. Of course, things weren't entirely straightforward – it seems they never are! – but Rod Burstall and I succeeded in designing the Clear specification language [8, 9, 10], which integrated initial and loose semantics with generic modules using "data constraints"[7] by extending an idea of Horst Reichel [161]. Clear seems to have been the first specification language with a rigorous semantic definition, and its modules seemed promising as a way to handle large systems.

I badly wanted to execute specifications, because I had noticed that it is all too easy to write them incorrectly, and I also thought it could be very helpful in teaching. Around 1974, I conceived the OBJ language for this purpose, using order sorted algebra[8] with mixfix syntax, and with term rewriting as operational semantics [42]; the goal was to make specifications as readable and testable as possible. The final OBJ3[9] [114] version of OBJ was implemented by a team led by José Meseguer, including contributions by Kokichi Futatsugi, Jean-Pierre Jouannaud, Claude and Hélène Kirchner, David Plaisted, Joseph Tardo, and others [104, 100, 27, 114]; this system provided both loose and initial semantics, rewriting modulo equations, generic modules, order sorted algebra with retracts, and user definable builtins[10]; OBJ2 was heavily used in designing OBJ3 [125], and I think greatly speeded up this effort, by facilitating team communication and documenting interfaces. Many other languages have followed OBJ's lead, including ACT ONE [24], which was used in the well known LOTOS hardware description language.

Although I never thought program verification had much practical value, I do think it has educational value, and in 1996, Grant Malcolm and I published a book on verifying imperative programs using OBJ3, based on a course we taught at Oxford [88]. Unlike other books on this topic, every program proof in the book is executable, and students can do all their homework on a computer; this produced a large improvement in both motivation and understanding, presumably in part due to the addictive quality of programming [141]. The use of algebra avoids the awkwardness and/or lack of rigor of techniques like predicate transformers and three valued logic that are found in some other books on this subject.

It is unhealthy to confuse a formal notation with a formal method. A *method* should say how to do something, whereas a *notation* allows one to say something [141]; thus OBJ is a notation, but using it as described in [88] gives a method for proving properties of imperative programs. Methods are rarely completely mechanical, because some of the problems that must be faced are usually uncomputable (e.g., finding loop invariants). Nevertheless, the method of [88] is surprisingly effective, in part because OBJ does all the routine work mechanically, and even provides hints to help with doing much of the non-routine work. For me, the frontier of research in this area is the use of systems like OBJ3 for theorem proving, e.g., in first order logic with equalities as atoms [71], or for verifying distributed concurrent systems (see [89] and see Section 2.10).

OBJ later developed a whole family of extensions, some of which are discussed in Section 2.6, and then

[7] It is interesting to notice that these must be *morphisms*, rather than just theory inclusions.

[8] Actually, a precursor called error algebra, motivated by the importance of error handling in real systems.

[9] "OBJ" refers to the general design, while "OBJ3" refers to a specific implementation.

[10] These were originally intended for providing builtin data structures like numbers, but were later used in implementing complex systems on top of OBJ, since they allow access to the underlying Lisp system [114].

it spawned a next generation, which is even now under construction. Some of the most important members of this next generation are the following:

1. SpecWare is a product from Kestrel Institute that has been very successful in synthesizing a wide range of scheduling algorithms, some of which are in daily practical use. SpecWare has a top level command `colimit` which computes the colimit of a collection of theories and theory morphisms [165].
2. Maude [148] is an extension of OBJ to rewriting logic [147], which is particularly suited to specifying concurrency. This project is led by Dr. José Meseguer at SRI International, where most of the original OBJ3 development was done, with support from the Office of Naval Research. Maude also has an efficient implementation and a number of interesting new features, including a logical and operational foundation in membership equational logic [149].
3. The CafeOBJ project [29] aims to make algebraic formal methods accessible to practicing software engineers. The CafeOBJ language is similar to OBJ, but enriched with features for both rewriting logic (as in Maude) and hidden algebra (see Section 2.10), to help specify and verify distributed concurrent systems. The CafeOBJ consortium includes several large Japanese companies, and is supported by MITI (the Japanese Ministry of Industry and Technology); more information on this project can be obtained from `http://ldl-www.jaist.ac.jp:8080/cafeobj` and from [20].
4. CoFI is another large effort to design and build an algebraic specification language. It is a highly collaborative multinational project with a distinctively European flavor, much influenced by the success of OBJ; see `http://www.brics.dk/Projects/CoFI`.

The most recent information on the OBJ family of systems and its relatives can be obtained from the website `http://www.cs.ucsd.edu/goguen/sys/obj.html`.

The motivation for all this work is of course to provide tools to formalize and verify the meaning of software and hence improve programming practice. From a purely theoretical point of view, there has been, and still is, a great deal of progress; but this has only made it more painfully clear that social issues play a dominant role in the transition to practical applications. There is an old saying that if you invent a better mouse trap, the world will beat a path to your door. But it's not true. You need to do field testing, file an environmental impact statement, get a designer label, mount a large advertising campaign – and then you need to train the mice!

2.5 Institutions

Because Clear is based on colimits of theories, Burstall and I were able to give it a very general semantics independent of the underlying logic in which theories are expressed, provided that logic has certain simple and very usual properties, which constitute the notion of **institution** [73, 75]. The basic feature of institutions is a duality between models and the (logical) sentences used in specifications arising through a relation of satisfaction that is parameterized by the signature involved. In the traditional cases of equational and first order logic, models are algebras and first order structures, respectively, but for LILEANNA (see Section 2.6), models are given by Ada programs. So institutions give a way to deal with issues in programming and specification languages (as well as databases and other kinds of system) independently of their underlying logic; I really love this kind of generality.

The theory of institutions was developed further in [74], showing how to generate institutions from the simpler structures of charters and parchments, how to put institutions together, and how to greatly generalize them; in particular, morphisms of sentences are introduced to support proof theory. The notion of inclusion system was introduced in [21] to axiomatize the notion of inclusion morphism in categorical terms, and then used to study some mathematical properties of specification modules, including the relation between pushouts preserving (various kinds of) conservative extension, Craig style interpolation properties, and some distributive laws for information hiding. There is now a rather large literature on institutions, with applications to many different areas, e.g., [17] concerns multi-institutional specification. However, it did take nine years (!) for the basic paper on institutions to be published in journal form [75]; this is the longest refereeing and editorial delay of which I ever heard.

2.6 Parameterized Programming and Generic Modules

Parameterized programming [48, 52, 27, 28] makes the advantages of the Clear module system available for real programming languages, as well as for more practical specification languages. In addition to semantic interfaces for generic modules (where axioms describe when the module will behave as advertised), parameterized programming provides module expressions to describe systems as interconnections, and views both to describe module bindings for instanting generics, and to serve as global assertions about semantic properties of subsystems; default views greatly simplify module expressions, and multiple inheritance for modules arises in a natural way. These ideas were first implemented in OBJ3. Although I'm happy that some of this influenced the languages Ada, ML, and C++, it can still be distressing to see the compromises involved.

LIL [50] extends parameterized programming to handle programs and specs together, by giving each module a specification "header" as well as implementations. LIL provides "two dimensional" module composition following the "CAT" ideas [72], where vertical structure refers to the layering of software to use capabilities from lower layers, while horizontal structure refers to a single layer. LIL has been implemented as LILEANNA [168, 169], which uses Ada for code and ANNA [135] for specs; it has been used to build helicopter navigation software. New features of LILEANNA include operations to add, delete and modify module functionality, at both the code and spec levels. A formal semantics is given for all this in [111], using a concrete set theoretic exposition of institutions; some general "laws of software engineering" are given showing how various module operations relate. Hyperprogramming [55] extends the idea of organizing information around a specification header to support requirements as well as specs and code, with traceability, controlled evolution, and management of configurations, versions, families, documentation, etc., as well as system generation and software reuse. All this is of course intended to ease the development of large systems, and in particular, to make reuse more effective in practice.

An approach to software architecture based on these ideas is given in [67], where module expressions provide a module connection (also called an architecture description) language. Any mixture of architectural styles can be supported, and modules can involve information hiding. Detailed design and coding are unnecessary if a suitable database of specifications and relationships among them is available, because executing a module expression yields an executable system, constructed by manipulating and linking implementation modules. The underlying ADT of the database is called a **module graph**; it includes specs, source code, compiled code, and many kinds of relationship, mostly among specs. So far there has been little interest in these ideas and even some antagonism; perhaps the community is too fragmented to accept a combination of formal semantics, mixed architectural styles, and generating systems from designs.

2.7 Order Sorted Algebra

Real software has many features that are difficult or impossible to treat with ordinary many sorted algebra. These include the raising and handling of exceptions, overloaded operators, subtypes, inheritance, coercions, and multiple representations. Error algebra [42] was a first crack at some of these problems, although it didn't work out. The second try was order sorted algebra (hereafter, **OSA**) [44], which reached fruition in joint work with Meseguer [99, 151]. This approach provides a partial ordering relation on sorts, interpreted semantically as subset inclusion among model carriers. Meseguer and I proved [151] that many simple ADTs have no adequate many sorted equational specification, because (what we call) the constructor-selector problem can't be solved in this setting. OSA is only slightly more difficult than many sorted algebra, and essentially all results generalize without much fuss; in particular, there are initial models and an efficient operational semantics. Because OSA is strongly typed, many terms that intuitively should be well formed because they evaluate to well formed terms, are actually ill formed; [99] introduced *retracts* to handle this problem. Sort constraints [99] extend OSA to support equational definitions of bounded data structures and partial operations. There are now many different variants and extensions of OSA, too numerous to mention here, although Meseguer's membership equational logic [149] should not be omitted. Although successful in this sense within the algebraic specification community, OSA seems to have had little influence elsewhere, and retracts have not been taken up anywhere.

2.8 Semantics of Logic Programming

Eqlog [95, 15] combines (equality based) functional programming with (predicate based) logic programming, by combining their logics, which are equational and Horn clause logic respectively, to get Horn clause logic with equality (actually the order sorted version). Eqlog's operational semantics also combines those of functional and logic programming, using both term rewriting and unification with backtracking; moreover, the design permits efficient special purpose algorithms for builtin types like numbers and lists, as well as narrowing to solve equations over user defined ADTs. Eqlog introduced several features that were new for logic programming, including user definable ADTs, strong typing with subsorts and overloading, generic modules, and a "wide spectrum" integration of coding with specification, design and prototyping. A good deal of theory was done to support Eqlog, including: complete rules of deduction, and Herbrand and initiality theorems, both for order sorted Horn clause logic with equality [79, 96]; order sorted unification [152]; order sorted narrowing and resolution [79, 152]; correctness criteria for builtin algorithms; and an initial model semantics generalizing and subsuming the traditional Herbrand universe construction. I still can't understand why the logic programming community prefers fixpoint semantics to this elegant algebraic approach.

2.9 Logical and Multi-Paradigm Programming

Major programming styles or "paradigms" that have emerged in addition to traditional imperative programming include functional programming, logic programming, and object oriented programming. Each can be considered a kind of logical programming [49], where a *logical programming language* is characterized as follows:

- its statements are sentences in some logic;
- its computation is deduction in that logic; and
- its denotational semantics is given by models in the logic.

For example, higher order functional programming is (or can be) based on higher order equational logic, and OBJ is based on order sorted first-order conditional equational logic. Similarly, logic programming is based on Horn clause logic. This approach can be made precise using institutions [96, 74, 49], and it has been enriched and extended by Meseguer with his theory of "general logics" [146].

In this setting, it is natural to generalize initial algebra semantics to *initial model semantics* [95, 96], using initiality in an arbitrary category; this often yields simpler proofs of general results by avoiding details of construction and representation. The initial model idea appears in a new guise in [96], to handle the semantics of logic programming over builtin types and algorithms as *free extensions* of the given builtin model. This idea also features in the elegant semantics for so called *constraint based* programming developed by Diaconescu [15, 16]. However initiality is not the right semantics for every logical language.

Programming paradigms can be combined by viewing them as logical programming languages and then combining their logics [49, 74, 96]. Thus Eqlog combines functional and logic programming [95, 96], and FOOPS combines object oriented and functional programming, by using reflective order sorted conditional equational logic [97, 115], since features of the object paradigm can be obtained by reflectivizing other logics; FOOPlog combines all three paradigms [97] by reflectivizing Horn clause logic with equality. All implementations of these systems were built on top of OBJ3. This research direction went so far as designing and prototyping special purpose hardware, the Rewrite Rule Machine, for executing declarative languages efficiently, based on term rewriting chips [98, 133, 57]. Despite all the interest once shown in declarative programming, the Fifth Generation, etc., there seems to be little interest in precise explications for declarative and logical programming and reflection, or in general purpose declarative architectures. See [11] for the latest on reflective logic and its applications.

2.10 Hidden Algebra

While (order sorted) equational logic works well for unchanging (immutable or "Platonic") data types like the numbers, it can be awkward for software modules having an internal state that changes over time. In

1982, Meseguer and I developed a theory of abstract machines [92] for this purpose, and proved minimal (final) and initial realization theorems for them; this theory naturally generalizes algebraic ADTs as well as classical automata. The minimal realization adjunction for automata [38] helped inspire this work, and it was also pleasant to realize that many intuitions from John Guttag's early work could be vindicated [116, 117].

In collaboration with Răzvan Diaconescu, Rod Burstall, and most recently especially Grant Malcolm, this work has developed into a new *hidden algebra* approach [54, 58, 77, 7, 87, 140, 89], intended to facilitate proving properties of *designs*, as opposed to code, and in particular, to facilitate refinement proofs, that one level of design is correctly realized by another. The main contribution is *hidden coinduction* techniques for (relatively) easy proofs of *behavioral properties*; this is important for software engineering because many practical implementation techniques provide the desired behavior, without realizing it as in the specification; hidden theories capture what have elsewhere been called behavioral types. This constitutes a *method* for using notations like OBJ3 and CafeOBJ to verify software designs; it is especially suitable for the object paradigm. The distinction between hidden and visible sorts allows the latter to be used for immutable data types (typically given by initial semantics). Hidden algebra has also opened intriguing new perspectives on nondeterminism and concurrency: nondeterminism arises naturally simply by *not specifying* some behaviors [89]; and concurrency is described by an elegant universal[11] construction on hidden theories [77]. A hidden Herbrand theorem which unifies the object and logic paradigms at the logical level is proved in [90]. There is now an excellent hidden group at Oxford, including Grant Malcolm, Corina Cîrstea, James Worrell, and Simone Veglioni [139, 13, 14, 172]; the recent proof that categories of hidden algebras are topoi [142] is especially exciting. There is also an exciting flurry of hidden activity around the CafeOBJ project at the Japan Institute of Science and Technology, which includes Răzvan Diaconescu, Kokichi Futatsugi, Shusaku Iida, Dorel Lucanu and Michihiro Matsumoto [18, 19, 20, 121].

2.11 Distributed Cooperative Engineering with TATAMI

Typical software engineering projects have multiple workers with multiple tasks that interleave in complex ways, often working at multiple sites with different schedules, so that it is difficult to share information and coordinate tasks; documentation is often hard to find, out of date, incomplete, or non-existent; requirements change; specifications change; personnel change; and it is hard to determine which parts of the system most need attention (e.g., see [137], especially its hypergraph model of evolution). Despite all this, most formal methods and tools to support them take a single user with a single unchanging task as their (usually implicit) model of interaction.

Recent work at the UCSD Meaning and Computation Lab seeks to address the distributed cooperative aspect of software engineering with an environment called TATAMI [80, 102] for CafeOBJ, having the capability (but *not* the necessity) for complete formal verification. Formality provides a discipline for both designing TATAMI and using it; however, the most practical use exploits the task structure of formality without requiring logical completeness, to ensure that all relevant dependencies are known, and that documentation, test cases, etc. are in predictable locations. Confidence values in the unit interval are associated with project tasks instead of Boolean truth values; this fuzzy logic (following [33]) allows critical path analysis to aid task allocation, taking account of different levels of formality and criticality.

TATAMI is supported by a truth maintenance protocol that resolves inconsistencies while updating local databases, allowing multiple versions at multiple sites, including incomplete and even incorrect proofs. The Kumo[12] tool generates websites for project documentation and assists with verification; it is now being used to populate a library with tutorial examples; see `http://www.cs.ucsd.edu/groups/tatami`. The output of Kumo can be read by any web browser, and in addition to the proof itself, provides for proof execution on remote servers, as well as animation and informal explanations. User interface design for this system uses algebraic semiotics (see Section 3.3), and its requirements were driven by my own participant observation as a software engineer. We are also using ideas from narratology (the study of stories) and even cinema to organize project websites to facilitate navigation and comprehension; since TATAMI supports multiple

[11] In the sense of category theory; i.e., it is a construction defined purely in terms of morphisms.

[12] This is a Japanese word for spider.

development lines, it also supports multiple narrative lines [101]. This group includes Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Lin is the heroic implementer of the Kumo system, and Roşu has given a hidden algebraic correctness proof for the TATAMI protocol.

Preliminary experiments using Kumo and TATAMI have been encouraging. This project is struggling to make earlier theoretical work more relevant to practice; however it is well beyond the capability of one small research group to build a truly industrial strength environment. So far this research has been greeted largely with incomprehension.

2.12 Discussion

Although my early work may have an austere kind of beauty from its abstraction and generality, it seems difficult for engineers to integrate such results into practice. Much of my later research has tried to understand what is required to bridge this divide, and in particular, my excursions into the social, described in the next section, have this motivation.

Looking back over Section 2 above, the extent to which I have failed to rise above the rather partisan divisions that are so much part of the present research scene is embarrassing, and has compromised my goal of evaluating ideas from a broad social perspective. I can only say that I have done my best at this time, and hope to have more perspective in the future.

3 Some Social Aspects of Computing

As already noted several times, social issues can dominate computer technology, e.g., in the construction of real systems that must be used in some social context, and in the propagation of new ideas into practice. This section reviews some approaches to reconciling computer technology with its social context. Sometimes algebraic methods are applied to the social, and sometimes social aspects of computation theory are considered.

3.1 Discourse Analysis

My earliest adventures into the social sciences were in discourse analysis, an area of sociolinguistics concerned with the large grain structure of language. Several types of discourse have a definite regular structure, including planning [134], explanation [113] and command and control discourse [82, 85]. The latter was developed in studies supported by NASA, on statistical properties of aviation discourse in emergency situations, for application to aviation safety. Later work on multi-media instruction supported by the Office of Naval Research involved naturalistic experiments and ideas from semiotics, for application to human-computer interface design [84]; this later led to the work discussed in Section 3.3. Two other discourse types are stories [126] and jokes [162], which are interesting because they embed values of the speaker and audience, and can therefore be used to study those values [69, 86]. This was applied in [81] as part of a study to determine requirements for a system to computerize a small headhunting firm, by collecting stories and jokes told during breaks and at lunch, and then collating them into a "value system tree" for the firm, as described in [60, 86]; this work also showed how to extract dataflow diagrams from task oriented discourse. For some reason, linguistics seems stuck on the syntax of sentences, despite the fact that there are important applications at higher levels. On the other hand, it must be admitted that there are a great variety of approaches to discourse analysis, the procedures tend to be time consuming, and at least the techniques used in my own work to describe discourse structure involve an unfamiliar formalism. So this work remains largely unknown. A critical overview of techniques for gathering information about social situations is given in [83]; in general, the greater the accuracy, the greater the difficulty.

3.2 Requirements Engineering

Case studies and experience suggest that flawed requirements may be the most significant source of errors in system development; moreover, it has been shown that requirements errors are the most expensive to correct at later stages [5]. Case studies and experience also suggest that social, political and cultural factors are very often responsible for the flaws in requirements; however, this area has been little studied. It follows that studying social aspects of requirements engineering has great leverage. But (using our ongoing metaphor) requirements engineering involves problems deep down inside the great divide.

The Centre for Requirements and Foundations was founded in 1991 at Oxford University to work towards a scientific basis for requirements engineering taking adequate account of social issues, as well as advanced technology; another goal was to develop appropriate new methods and tools for capturing and analyzing requirements, to help build systems that are better for users, as well as less stressful for those involved in the building process. When I left Oxford, the Centre had completed two main projects, supported by a large grant from BT. The first was a case study in requirements elicitation using techniques from sociology and linguistics, particularly interaction (video) analysis, but also discourse and conversation analyses [118, 136]. The second was a classification of methods and tools used for requirements [4]. The first project built on early work with Linde on requirements elicitation [81], which was followed by a critical survey of elicitation methods [83]. The second project was in part inspired by work of Lyotard [138] on post-modernism. Other work from the Centre included a book [124] consisting of (revised) papers from a workshop organized by the Centre, some more papers [64, 123], and the TOOR object oriented tool for tracing requirements [160]. The view that the essence of requirements engineering is to reconcile social and technical aspects of system design was proposed in [65] and elaborated in later work [66, 69]; it amounts to saying that requirements engineering consists of building bridges across the divide. Hence there is no disciplinary home for this area, and thus despite its great economic importance, there is little academic effort devoted to it; in particular, I don't know of any degree programs in this area.

3.3 Algebraic Semiotics

Semiotics is defined as the theory of signs and their meanings, a very difficult area deep inside the social side of the great divide. Unfortunately there seem to be at least as many approaches to semiotics as there are authors who have considered it, and many other fields with different names also cover the same or closely related ground, e.g., cognitive linguistics [127, 26, 170]. Algebraic semiotics [70, 68] is one more: it tries to combine insights from both sides of the divide, to obtain a precise formulation of certain problems about meaning and to allow the construction of supporting technology. Among the main insights are: (1) signs mediate meaning (emphasized by Charles Saunders Peirce [157]); (2) signs come in structured systems (made clear and studied in detail for language by Ferdinand de Saussure [164]); (3) structure preserving maps ("morphisms") are often at least as important as structures[13]; and (4) discrete abstract structures can be described by algebraic theories (see Section 2.3). In algebraic semiotics, *sign systems* are algebraic theories with extra structure, and *semiotic morphisms* are used to study representations, metaphors, translations, meanings, etc.; because these map signs in one system to signs in another, rather than mapping individual signs, Saussure's insight is raised from sign systems to their morphisms. In [70, 68], techniques are also given for comparing the quality of semiotic morphisms, and a new version of categorical colimits, developed in collaboration with Grigore Roşu, is used for combining meanings and for studying the effect of context on meaning; this includes "blends" in the sense of [26]. The potential to connect diverse areas on both sides of the great divide and to enter new application areas, such as user interface design, seems very exciting.

When applied to user interface design, algebraic semiotics can model the content and structure of information through its representation [68, 80, 101, 102]; it is now being used to design interfaces for the TATAMI system (see Section 2.11). Although traditional ergonomics, HCI (human computer interface), and cognitive science are very good for issues like keyboard layout, color choice, font size, and window layout, they are less useful for more semantic problems. Examples using algebraic semiotics are accessible over the web at `http://www.cs.ucsd.edu/groups/tatami`, and a "semiotic zoo" of ordinary design choices that are bad for interesting reasons is available at `http://www.cs.ucsd.edu/users/goguen/zoo`, with algebraic

[13] This is an insight from mathematics. The journey through this paper has already encountered several cases where morphisms are important: initial extensions for constraint programming (in Section 2.9); data constraints (in Section 2.4); and inclusion systems (Section 2.4). Eilenberg and Mac Lane [25] gave this insight a more definite and systematic form with the invention of category theory.

explanations. The zoo illustrates how algebraic semiotics can be applied to syntax, understanding stories, and much more.

3.4 Fuzzy Logic and Information Theory

Many people have had the intuition that mathematical logic fails to capture the imprecision and robustness of practical reasoning. Fuzzy sets and logic [174] are a successful move in this direction, though it is far from capturing the richness and complexity of ordinary human reasoning. In the late 1960s, it was fashionable to give axioms on the category of things having some structure (with structure preserving morphisms) to characterize that category up to natural equivalence. This is what I did for fuzzy sets in my thesis [32, 34, 39]; earlier papers concerned other aspects of fuzzy sets, e.g., extending them to more general values than the unit interval [31, 33]. This was the first foundational work on fuzzy sets and fuzzy logic. Later I set up the Fuzzy Robot Users Group at UCLA, and did some early empirical work on fuzzy algorithms, though it was never properly published (but see [103] for an abstract). Some applications to philosophy and the social sciences, and some limitations of fuzzy set theory were discussed in [45]. Today fuzzy sets and fuzzy logic are very popular areas. But in the late 1960s, work in this area was bitterly opposed by more traditional parts of engineering, such as classical control theory, to the extent that I found it impossible to get funding to continue my research in this area, and had to abandon it.

It is (or should be) a scandal that in the middle of a period called the "information age" and characterized by an astonishing expansion of information technology, there is no adequate theory of information, nor even any adequate definition of information. In the late 1960s, I taught a course at the University of Chicago on traditional Shannon-style information theory [76], and encountered great difficulty trying to extend it to human situations; in fact this theory does not apply to meaning, but rather to data compression and transmission – after all, it was developed for the (then) Bell Telephone Company – because it ignores the crucial human aspects that underlie meaning. This motivated using categorical GST for a complexity based information theory, which was then applied to music [43]; here the information content of a behavior is the minimum value of the sum of weights of (hierarchical) interconnected components whose behavior is the given one. It was startling that the classical equalities and inequalities of information theory still hold in this exceedingly general setting; moreover it did capture some insights about the structure of music. Several Oxford students did experiments in this area for their MSc theses, but again it became clear that no purely formal approach, however abstract and general, could deal with human meaning in any deep sense [63] (see also [45]).

Recently I returned to this area, but from the opposite side of the great divide, defining information in social terms [69]: *An item of information is an interpretation of a configuration of signs for which members of some social group are accountable.* The goal is to get a theory of information adequate for understanding and designing systems that process information. This research draws on ideas from ethnomethodology [163], semiotics, logic, and the sociology of science. The paper [69] also presents some case studies and makes the perhaps surprising argument that because of its social situatedness, information has an intrinsic ethical dimension.

4 Philosophy of Computing

It is usually thought that philosophy has little to do with the practical engineering concerns that dominate computer science today. But philosophies are just coherent expositions of particular approaches to defining, approaching and (maybe) solving problems[14]. Nothing is more practical than a good philosophy: our approach to a situation has an effect, often decisive, on what happens. As Wesley Phoa [159] puts it,

> All practical work is based on philosophical presuppositions: they may be conscious or unconscious, innocuous or fatal. ... In any case, we might as well be aware of them, and aware of the alternatives. There is a more positive reason to be interested in philosophy though: philosophical reflection can be a potential source of practical ideas.

[14]Actually, even the notion that "problems" exist and should be "solved" is a philosophical presupposition that is open to question!

This section discusses some philosophies that seem especially relevant to today's computer science.

4.1 Realism and Idealism

Realism is the view that certain things (e.g., numbers) really exist; one can be "realistic" in this sense about many different things. For pure mathematics, realism is called *Platonism*. Here it seems harmless, perhaps even charming; but it can become dangerous when extended further into applied mathematics. Within computer science, systems analysis, systems engineering, requirements engineering, etc., realism can be the view that there really is such a "thing" as "the system," and (going beyond realism into confusion) that some particular model built for some particular purpose completely "captures" the system. This easily can (and often does) lead to incorrect decisions based on some weak area of the model, that should have been reevaluated and strengthened, but was not because of the erroneous belief that the model *really is* the system. I encountered some examples of this in the area of computer security, in connection with the so-called Bell-LaPadula security model [3, 91, 93].

It is easier to understand the attraction of realism when it is considered in the context of its opposition to *idealism*, the view that things (either some particular class of things, or in a more extreme form, all things) only exist in our minds; the most extreme form of this view is *solipsism*, that nothing exists outside oneself. The problem with idealism is that it seems to negate science (as well as religion and all metaphysics); so realists can be seen as trying to save "reality" for science (and/or religion).

My own view is that this is a false duality, based on an unexamined presupposition, namely dualism. Simplifying quite a bit, *dualism* asserts a rigid separation between a material realm and a spiritual realm; such a view was advocated by Descartes in order to gain a ground for science that was free from interference by religion. Today dualism is largely considered untenable, for example because there is no empirical evidence for a separate realm of the spiritual. So within this historical context, idealism can be seen as the attempt to reduce everything to the spiritual (though not usually in the religious sense) and realism as the attempt to reduce everything to the material – which is what most science is about anyway. In my view, both are crude attempts to eliminate Cartesian duality, which after all does reflect some aspects of everyone's experience. Instead of trying to collapse the duality to one of its poles, it seems preferable to *transcend* duality in a way that denies the actual existence of two realms, and still adequately explains our experience. My preference is for *phenomenology*, which starts with experience, disallowing any external distinctions; it then proceeds with a careful analysis of experience and its ground; however, this paper isn't the place to consider such issues; some further discussion and references may be found in [63].

4.2 Modernism

Modernism can be defined as belief in the adequacy of hierarchy, formalization and control to achieve desired ends without error. The success of computer science, along with technology as a whole, has long been seen as establishing the correctness of such a philosophy, so much so that it is usually not even explicitly formulated. But increasing experience with ever more ambitious projects has led to more awareness of the limitations of modernism. A classic example in software engineering is the so-called waterfall model of software development, which requires a rigid top-down hierarchical control of the development process, by dividing it into strictly sequential stages, typically something like requirements, design, specification, build and test. But such an organization of the work process makes it very difficult to correct errors in earlier stages, to learn from errors, and to respond to the plethora of changes in the surrounding context of technology, organization, law, etc. that are inevitable and unending in today's fast paced environment. Of course, now there are many process models that improve upon the waterfall model, but I think there is still insufficient appreciation of the importance of alternative techniques like rapid prototyping and user participation in design.

Theoretical computer scientists need not look so far afield for examples of excessive modernism. What might be called the "error free" school of formal methods aims for programs that have no bugs at all. For example, [22] claims

> we have ... "a calculus" for a formal discipline – a set of rules – such that, if applied successfully:

(1) it will have derived a correct program; and (2) it will tell us that we have reached such a goal.

From a narrow point of view, this effort succeeded, modulo certain technical difficulties which can however be corrected[15]. However, there is a fundamental difficulty, namely that it attempts to control the programming process by imposing a rigid top-down derivation sequence, working backwards from the initial top level specification (the "postcondition") to the final code, in which each step is derived by applying a "weakest precondition" (hereafter, "wp") formula.

It is not coincidental that the "wp calculus" requires significant human invention at exactly the most difficult points, namely the loops. And for most programs that go much beyond the trivial, the insights needed to write the loop invariants are tantamount to already knowing how to write the program; moreover, these insights are more difficult to achieve when using wp than they would be in a more conventional setting. When I was at Oxford, I saw several very good students who had been taught that the wp calculus was the right way to program, become so discouraged over the difficulties they experienced, that they came to believe they could never learn how to program and should therefore seek a different profession. This is a great pity! In general, rigid top-down ideologies inhibit experimentation and make it hard to explore tradeoffs. Moreover, they can be harmful to students, wasteful of time, reinforcing of an inflexible view of life, and inhibiting to intuition, learning and creativity. Finally, as noted in the first paragraph of Section 2.4, correctness of code is the wrong problem to solve. (An overview of some recent debates on philosophical foundations of formal methods may be found in [1].)

A further difficulty with formal methods is that they tend to be very brittle, in the sense that slight changes in a specification can lead to drastic changes in what must be done to achieve that spec. We can see the importance of this difficulty by noting that the very rapid rate of change of requirements, which is so typical of large projects, implies an even more rapid rate of change for specifications. This makes many formal methods very difficult, perhaps even impossible, to apply in practice [137]. Let me be clear that I am not criticizing formal methods as such – in fact, I believe they can be very useful in practice, *especially* for large programs, by focusing on the large grain structure of software (see Section 2.6 and [137]).

Jean-François Lyotard [138] defines *modernism* more broadly than at the beginning of this section, as any approach that justifies its claims to universality through a "grand unifying" story, which he calls a *meta-narrative*. For example, the realist approach to systems theory discussed in Section 4.1 tells the story of unique pre-existing systems really out there to be captured. By contrast, Lyotard says *postmodernism* is characterized by multiple "local language games" that cannot necessarily be unified, or even neatly classified (this notion of language game comes from late work of Wittgenstein). A great deal has been written about postmodernism, and though much of it is trash, there is a general agreement that we may now be entering a postmodern period.

We have already seen the issue of whether there are many or just one language game in the discussion of requirements engineering in Section 3.2; my view is that requirements engineers must deal with multiple "stakeholders" who may have not just different viewpoints on the "same system," but may actually have completely different ways to conceptualize their experience. A related philosophical approach is the *deconstructionism*[16] of Derrida, which can be seen as a strategy for undermining meta-narratives. Phoa [159] applies deconstruction to aspects of theorem proving, software engineering and artificial intelligence; in particular, Phoa's view of requirements is remarkably similar to that presented here. It can also be argued that the approach to distributed cooperative theorem proving and software engineering taken in the TATAMI system (Section 2.11) is postmodern, or even Derridean, because it supports multiple points of view, including views that are mutually inconsistent, or even self inconsistent, and because it uses multiple narratives as an organizational principle.

[15] These include the following: (1) there is a gap in the logical foundations, in that the first order logic used for expressing conditions is not actually sufficiently expressive – something like the infinitary logic proposed by Erwin Engeler in the 1960s for this purpose is necessary; (2) many important programming features are not treated, including procedures, blocks, modules, and objects – in general, all large grain features are omitted; and (3) data structures and types are treated loosely, and variables that range over different kinds of item are not carefully distinguished.

[16] See also the hilarious Woody Allen film *Deconstructing Harry.*

4.3 Autopoiesis

Maturana and Varela [144] define an *autopoietic system* to be

> ... a network of processes of production of components that produces the components that: (i) through their interactions and transformations continuously regenerate the network of processes that produced them; and (ii) constitute it as a concrete unity in the space in which they exist by specifying the topological domain of its realization as such a network.

(See [2, 167, 143, 145] for more on this area, and see [171, 112] for some possibly ill-advised attempts to formalize this notion; also cf. footnote 4. Sorry for the confusing prose in this quote.)

The relevance of this to software engineering is discussed in [42]: Anyone familiar with large software projects knows that there is a sense in which they "have a life of their own," in that some projects seem healthy and vibrant from the start, and overcome even unexpected obstacles with enthusiasm and intelligence, while others always seem disorganized and depressed, suffering from such symptoms as unrealistic goals, inadequate equipment, poor planning, (seemingly) insufficient funding, faulty communication, indecisive leadership, frequent reorganizations, and/or deep rifts between internal factions.

A software development project is not a formal mathematical entity. Perhaps it is usefully seen as an autopoietic process, an evolving organization of informational structures, continually recreating itself by building, modifying, and reusing its structures; in the language of Maturana, this is "development through mutually recursive interactions among structurally plastic systems" [143]. For example, an unhealthy project may struggle for survival by reassigning responsibilities, redefining subprojects, and even trying to reconstrue the conditions that define its success. On the other hand, a healthy project may develop new tools to enhance its productivity.

In this view, computers, printouts, compilers, editors, design tools, and even programmers, can be seen as supporting substrates, just as body parts are supporting substrates for a person[17]. Autopoietic systems are about as far as we know how to get from rigid top-down hierarchical goal-driven control systems; autopoietic systems thrive on error, and reconstruct themselves on the basis of what they learn from their mistakes. Autopoiesis can be considered an implementation technique for postmodernism. See [59] for more on attitudes towards errors in computer science.

4.4 Ethics

One aspect of the great divide that seems especially troubling in the late twentieth century is the separation of technology and science from ethics. A positive sign is the increasing availability of courses on professional ethics in engineering schools; on the other hand, ethical behavior seems to be on the decrease. Of course, there is a huge philosophical literature on ethics, but little of it directly addresses technology. My own concern is to bridge the gap between technology and ethics on intellectual grounds. In [69], I argue for an inherent ethical dimension to information (but not mere data), through its being embedded in a context of concern by some social group (see the definition of information in Section 3.4). This is not an appropriate place for details, but we should recall that understanding values can be crucial for getting requirements that match user needs. Other perspectives on the "great divide" can be found in the book [6] in which the paper [69] appears. A more radical view appears in [119], where Heidegger considers Western civilization to be fundamentally entangled with a separation of technology from ethics, based on an untenable instrumental conception of technology; see also the discussions in [61] and [63]. Heidegger [119] further claims that by questioning deeply enough into the essence of technology, perhaps in desperation, we may find what we need to go beyond the current impasse.

But no one should think that postmodernism, deconstructionism or phenomenology is going to answer every question concerning technology, the modern world, or the philosophy of computation. The issues that must be faced are extremely complex and diverse, and it appears to me that we are only at the beginning of some new ways of thinking. If our civilization survives, there will doubtless be many profound changes ahead in how we think about technology and its relation to society.

[17] Of course, this does not mean that groups have moral or spiritual priority over individuals, or that people should be viewed as components of systems in anything like the same way that Ada packages can be.

5 Some Concluding Remarks

I'm afraid that the reader may have found this paper rather a long strange trip[18], starting from the practice of software engineering, then going to category theory, and eventually ethics, passing through topics like equational deduction, various programming and specification paradigms, semiotics, theorem proving, requirements engineering and philosophy. Nevertheless, theoretical computer science has been a constant theme, for example appearing in the application of ADTs and theory morphisms to semiotics and user interface design, and in the philosophical exploration of difficulties that arise in computing practice. I believe that the kind of concept developed in theoretical computer science, especially its algebraic and categorical branches, is very well suited to combating the fragmentation of computer science as a whole, and hope to have given some salient examples; this seems especially important for education.

It seems valuable to use algebraic theories to model the kind of situated abstract data types that appear in semiotics, and it also seems valuable to study the trajectories of projects that involve theory. Both kinds of research straddle the great divide, and both are risky. It is much safer to stay within the confines of a single discipline, and if possible work on a single well defined problem of recognized practical importance. Few people want to read a paper like [68] in a serious way, because it requires familiarity with diverse areas of philosophy, mathematics, linguistics, literature and computer science; so the author isn't likely to get much recognition, and it will be hard to publish. But the safe way can also be a painful way. Working on a narrow well defined problem of recognized importance will almost guarantee intense competition. Of course this is what academic departments like, because it makes comparison and measurement easier. In this way, departmental boundaries are inimical to interdisciplinary work, as well as to innovative and/or very abstract work. On the other hand, a community is necessary to develop tools and methods and get them actually used; transfering either theory or technology is by definition a social process [130], so that no lone researcher or small group can hope to get very far. But this is not to say that fame and fortune are the only measures of success; on the contrary, quite different measures like aesthetics and coherence seem to be at least as important for theoretical computer science.

In this paper, I've tried to share some of the hopes and disappointments I've felt for various ideas and projects; these remarks are not intended as bragging or complaining; at this stage in my life, I don't take any of them seriously (though I must admit that some did seem very serious at the time). The goal of these remarks is to help younger researchers. Most work of most researchers never has very much influence, every career has difficulties, and innovative and/or interdisciplinary work is especially likely to be difficult. Moreover, although the conventions of scientific writing hide it, every researcher has some emotional involvement with his/her work, and this is something that should be dealt with, not just ignored. Those interested in the sociology of technology may also find it interesting to see how ideas have sometimes flowed from one area to another. Our civilization needs to heal the wound between the social and the technical-scientific world views, and I hope to have given some hints about how this might occur; see also the charming documentary novel *Aramis, or the Love of Technology*, by Bruno Latour [131].

After discussing this paper with a student, he asked if I felt "melancholy" looking back over a whole career of tossing algebraic flowers down the great divide. Not really, I replied; to be sure, this is a sad metaphor, but is not all life similar? Of course there are difficulties, and perhaps there are successes. But commitment to our art and craft, to our profession, to our own integrity, and to other people, are far more important; life occurs as it is lived, and to live it fully is to appreciate joys and pains as they are, rather than in the light of ambitions that can only intensify pain and confusion.

Acknowledgements

I enjoy working with others, and much of my favorite research has been joint work with Rod Burstall, Răzvan Diaconescu, Kokichi Futatsugi, Charlotte Linde, Grant Malcolm, José Meseguer, David Plaisted, James Thatcher, Will Tracz, Eric Wagner, Timothy Winkler, Jesse Wright, and personnel at the UCSD Meaning and Computation Lab, including Kai Lin, Eric Livingston, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Especially I thank Rod Burstall, Peter Landin, Saunders Mac Lane, John McCarthy, and Christopher

[18] With a nod of the head to the Grateful Dead song *Truckin'*.

Strachey, whose pioneering research inspired my interests in semantics, algebra and logic. Burstall and Mac Lane have done even more, they have encouraged and guided me, and my gratitude and debt are very great; I have also learned much from Charlotte Linde about sociolinguistics and discourse analysis. I wish to thank the organizations that have supported my research, including the National Science Foundation, the Office of Naval Research, the National Aeronautics and Space Administration, British Telecom, Fujitsu, several European Union projects, and two MITI (Japan) projects, including the CafeOBJ project.

References

[1] Jon Barwise. Mathematical proofs of computer system correctness. Technical Report CSLI-89-136, Center for the Study of Language and Information, Stanford University, August 1989.

[2] Gregory Bateson. *Mind and Nature: A Necessary Unity.* Bantam, 1980.

[3] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical report, MITRE, 1974.

[4] Matthew Bickerton and Jawed Siddiqi. The classification of requirements engineering methods. In Stephen Fickas and Anthony Finkelstein, editors, *Requirements Engineering '93*, pages 182–186. IEEE, 1993.

[5] Barry Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[6] Geoffrey Bowker, Leigh Star, William Turner, and Les Gasser. *Social Science, Technical Systems and Cooperative Work: Beyond the Great Divide.* Erlbaum, 1997.

[7] Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: an institutional approach. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 75–92. Prentice Hall, 1994.

[8] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[9] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[10] Rod Burstall and Joseph Goguen. An informal introduction to specifications using Clear. In Robert Boyer and J Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic, 1981. Reprinted in *Software Specification Techniques*, Narain Gehani and Andrew McGettrick, editors, Addison-Wesley, 1985, pages 363–390.

[11] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In José Meseguer, editor, *Proceedings, First International Workshop on Rewriting Logic and its Applications.* Elsevier Science, 1996. Volume 4, *Electronic Notes in Theoretical Computer Science.*

[12] Steven Collins. *Selfless Persons.* Cambridge, 1983.

[13] Corina Cîrstea. A semantic study of the object paradigm, 1996. Transfer thesis, Programming Research Group, Oxford University.

[14] Corina Cîrstea. Coalgebra semantics for hidden algebra: parameterized objects and inheritance, 1997. Paper presented at 12th Workshop on Algebraic Development Techniques.

[15] Răzvan Diaconescu. *Category-based Semantics for Equational and Constraint Logic Programming.* PhD thesis, Programming Research Group, Oxford University, 1994.

[16] Răzvan Diaconescu. A category-based equational logic semantics to constraint programming. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, pages 200–222. Springer, 1996. Lecture Notes in Computer Science, Volume 389.

[17] Răzvan Diaconescu. Extra theory morphisms for institutions: logical semantics for multi-paradigm languages. Technical Report IS-RR-96-0024S, Japan Institute of Science and Technology, 1996. To appear, *J. Applied Categorical Structures.*

[18] Răzvan Diaconescu. Foundations of behavioural specification in rewriting logic. In José Meseguer, editor, *Proceedings, First International Workshop on Rewriting Logic and its Applications*. Elsevier Science, 1996. Volume 4, *Electronic Notes in Theoretical Computer Science*.

[19] Răzvan Diaconescu and Kokichi Futatsugi. Logical semantics for CafeOBJ. Technical Report IS-RR-96-0024S, Japan Institute of Science and Technology, 1996.

[20] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6. World Scientific, 1998. To appear, AMAST Series in Computing.

[21] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993.

[22] Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the Association for Computing Machinery*, 18:453–457, 1975.

[23] Hans-Dieter Ehrich, Joseph Goguen, and Amilcar Sernadas. A categorial theory of objects as observed processes. In Jaco de Bakker, Willem de Roever, and Gregorz Rozenberg, editors, *Foundations of Object Oriented Languages*, pages 203–228. Springer, 1991. Lecture Notes in Computer Science, Volume 489.

[24] Hartmut Ehrig, Werner Fey, and Horst Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83–03, Technical University of Berlin, Fachbereich Informatik, 1983.

[25] Samuel Eilenberg and Saunders Mac Lane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58:231–294, 1945.

[26] Gilles Fauconnier and Mark Turner. Conceptual projection and middle spaces. Technical Report 9401, University of California at San Diego, 1994. Dept. of Cognitive Science.

[27] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[28] Kokichi Futatsugi, Joseph Goguen, José Meseguer, and Koji Okada. Parameterized programming in OBJ2. In Robert Balzer, editor, *Proceedings, Ninth International Conference on Software Engineering*, pages 51–60. IEEE Computer Society, March 1987.

[29] Kokichi Futatsugi and Ataru Nakagawa. An overview of Cafe specification environment. In *Proceedings, ICFEM'97*. University of Hiroshima, 1997.

[30] W. Wyat Gibbs. Software's chronic crisis. *Scientific American*, pages 72–81, September 1994.

[31] Joseph Goguen. L-fuzzy sets. *Journal of Mathematical Analysis and Applications*, 18(1):145–174, 1967.

[32] Joseph Goguen. *Categories of Fuzzy Sets*. PhD thesis, Department of Mathematics, University of California at Berkeley, 1968.

[33] Joseph Goguen. The logic of inexact concepts. *Synthese*, 19:325–373, 1968–69.

[34] Joseph Goguen. Categories of V-sets. *Bulletin of the American Mathematical Society*, 75(3):622–624, 1969.

[35] Joseph Goguen. Mathematical representation of hierarchically organized systems. In E. Attinger, editor, *Global Systems Dynamics*, pages 112–128. S. Karger, 1971.

[36] Joseph Goguen. Minimal realization of machines in closed categories. *Bulletin of the American Mathematical Society*, 78(5):777–783, 1972.

[37] Joseph Goguen. Categorical foundations for general systems theory. In F. Pichler and R. Trappl, editors, *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta Books, 1973.

[38] Joseph Goguen. Realization is universal. *Mathematical System Theory*, 6:359–374, 1973.

[39] Joseph Goguen. Concept representation in natural and artificial languages: Axioms, extensions and applications for fuzzy sets. *International Journal of Man-Machine Studies*, 6:513–561, 1974.

[40] Joseph Goguen. Objects. *International Journal of General Systems*, 1(4):237–243, 1975.

[41] Joseph Goguen. Semantics of computation. In Ernest Manes, editor, *Proceedings, First International Symposium on Category Theory Applied to Computation and Control*, pages 151–163. Springer, 1975. (San Fransisco, February 1974.) Lecture Notes in Computer Science, Volume 25.

[42] Joseph Goguen. Abstract errors for abstract data types. In Eric Neuhold, editor, *Proceedings, First IFIP Working Conference on Formal Description of Programming Concepts*, pages 21.1–21.32. MIT, 1977. Also in *Formal Description of Programming Concepts*, Peter Neuhold, Ed., North-Holland, pages 491–522, 1979.

[43] Joseph Goguen. Complexity of hierarchically organized systems and the structure of musical experiences. *International Journal of General Systems*, 3(4):237–251, 1977.

[44] Joseph Goguen. Order sorted algebra. Technical Report 14, UCLA Computer Science Department, 1978. Semantics and Theory of Computation Series.

[45] Joseph Goguen. Fuzzy sets and the social nature of truth. In M.M. Gupta and Ronald Yager, editors, *Advances in Fuzzy Set Theory and Applications*, pages 49–68. North-Holland, 1979.

[46] Joseph Goguen. Some ideas in algebraic semantics. In Ken Hirose, editor, *Mathematical Logic and Computer Science*. IBM Japan, 1979. Proceedings, Third IBM Symposium on Mathematical Foundations of Computer Science.

[47] Joseph Goguen. How to prove algebraic inductive hypotheses without induction, with applications to the correctness of data type representations. In Wolfgang Bibel and Robert Kowalski, editors, *Proceedings, Fifth Conference on Automated Deduction*, pages 356–373. Springer, 1980. Lecture Notes in Computer Science, Volume 87.

[48] Joseph Goguen. Parameterized programming. *Transactions on Software Engineering*, SE–10(5):528–543, September 1984.

[49] Joseph Goguen. One, none, a hundred thousand specification languages. In H.-J. Kugler, editor, *Information Processing '86*, pages 995–1003. Elsevier, 1986. Proceedings of 1986 IFIP Congress.

[50] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251–263, and in *Domain Analysis and Software Systems Modelling*, Rubén Prieto-Díaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125–137.

[51] Joseph Goguen. Memories of ADJ. *Bulletin of the European Association for Theoretical Computer Science*, 36:96–102, October 1989. Guest column in the 'Algebraic Specification Column.' Also in *Current Trends in Theoretical Computer Science: Essays and Tutorials*, World Scientific, 1993, pages 76–81.

[52] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[53] Joseph Goguen. What is unification? A categorical view of substitution, equation and solution. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989.

[54] Joseph Goguen. An algebraic approach to refinement. In Dines Bjorner, C.A.R. Hoare, and Hans Langmaack, editors, *Proceedings, VDM'90: VDM and Z – Formal Methods in Software Development*, pages 12–28. Springer, 1990. Lecture Notes in Computer Science, Volume 428.

[55] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology*. Joint System Development Corporation, Tokyo, Japan, January 1990.

[56] Joseph Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, March 1991.

[57] Joseph Goguen. Semantic specifications for the Rewrite Rule Machine. In Aki Yonezawa and Takayasu Ito, editors, *Concurrency: Theory, Language and Architecture*, pages 216–234, 1991. Proceedings of a U.K.–Japan Workshop; Springer, Lecture Notes in Computer Science, Volume 491.

[58] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[59] Joseph Goguen. The denial of error. In Christiane Floyd, Heinz Züllighoven, Reinhard Budde, and Reinhard Keil-Slawik, editors, *Software Development and Reality Construction*, pages 193–202. Springer Verlag, 1992.

[60] Joseph Goguen. The dry and the wet. In Eckhard Falkenberg, Colette Rolland, and El-Sayed Nasr-El-Dein El-Sayed, editors, *Information Systems Concepts*, pages 1–17. Elsevier North-Holland, 1992. Proceedings, IFIP Working Group 8.1 Conference (Alexandria, Egypt).

[61] Joseph Goguen. Hermeneutics and path. In Christiane Floyd, Heinz Züllighoven, Reinhard Budde, and Reinhard Keil-Slawik, editors, *Software Development and Reality Construction*, pages 39–44. Springer Verlag, 1992.

[62] Joseph Goguen. Sheaf semantics for concurrent interacting objects. *Mathematical Structures in Computer Science*, 11:159–191, 1992.

[63] Joseph Goguen. Truth and meaning beyond formalism. In Christiane Floyd, Heinz Züllighoven, Reinhard Budde, and Reinhard Keil-Slawik, editors, *Software Development and Reality Construction*, pages 353–362. Springer Verlag, 1992.

[64] Joseph Goguen. Social issues in requirements engineering. In Stephen Fickas and Anthony Finkelstein, editors, *Requirements Engineering '93*, pages 194–195. IEEE, 1993.

[65] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic, 1994.

[66] Joseph Goguen. Formality and informality in requirements engineering. In *Proceedings, International Conference on Requirements Engineering*, pages 102–108. IEEE Computer Society, April 1996.

[67] Joseph Goguen. Parameterized programming and software architecture. In *Proceedings, Reuse'96*, pages 2–11. IEEE Computer Society, April 1996.

[68] Joseph Goguen. Semiotic morphisms. Technical Report CS97–553, UCSD, Dept. Computer Science & Eng., 1997. Early version in *Proc., Conf. Intelligent Systems: A Semiotic Perspective, Vol. II*, ed. J. Albus, A. Meystel and R. Quintero, Nat. Inst. Science & Technology (Gaithersberg MD, 20–23 October 1996), pages 26–31.

[69] Joseph Goguen. Towards a social, ethical theory of information. In Geoffrey Bowker, Leigh Star, William Turner, and Les Gasser, editors, *Social Science, Technical Systems and Cooperative Work: Beyond the Great Divide*, pages 27–56. Erlbaum, 1997.

[70] Joseph Goguen. An introduction to algebraic semiotics, with applications to user interface design. In Chrystopher Nehaniv, editor, *Proceedings, Computation for Metaphors, Analogy and Agents*. University of Aizu, 1998. Aizu–Wakamatsu, Japan, 6–10 April 1998.

[71] Joseph Goguen. *Theorem Proving and Algebra*. MIT, to appear.

[72] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Report CSL-118, SRI Computer Science Lab, October 1980.

[73] Joseph Goguen and Rod Burstall. Introducing institutions. In Edmund Clarke and Dexter Kozen, editors, *Proceedings, Logics of Programming Workshop*, pages 221–256. Springer, 1984. Lecture Notes in Computer Science, Volume 164.

[74] Joseph Goguen and Rod Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In David Pitt, Samson Abramsky, Axel Poigné, and David Rydeheard, editors, *Proceedings, Conference on Category Theory and Computer Programming*, pages 313–333. Springer, 1986. Lecture Notes in Computer Science, Volume 240.

[75] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.

[76] Joseph Goguen and Lee Carlson. Axioms for discrimination information. *IEEE Transactions on Information Theory*, pages 572–574, September 1975.

[77] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.

[78] Joseph Goguen and Susanna Ginali. A categorical approach to general systems theory. In George Klir, editor, *Applied General Systems Research*, pages 257–270. Plenum, 1978.

[79] Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Operational semantics of order-sorted algebra. In Wilfried Brauer, editor, *Proceedings, 1985 International Conference on Automata, Languages and Programming*. Springer, 1985. Lecture Notes in Computer Science, Volume 194.

[80] Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Distributed cooperative formal methods tools. In Michael Lowry, editor, *Proceedings, Automated Software Engineering*, pages 55–62. IEEE, 1997. Lake Tahoe CA, 3–5 November 1997.

[81] Joseph Goguen and Charlotte Linde. Cost-benefit analysis of a proposed computer system. Technical report, Structural Semantics, 1978.

[82] Joseph Goguen and Charlotte Linde. Linguistic methodology for the analysis of aviation accidents. Technical report, Structural Semantics, December 1983. NASA Contractor Report 3741, Ames Research Center.

[83] Joseph Goguen and Charlotte Linde. Techniques for requirements elicitation. In Stephen Fickas and Anthony Finkelstein, editors, *Requirements Engineering '93*, pages 152–164. IEEE, 1993. Reprinted in *Software Requirements Engineering (Second Edition)*, ed. Richard Thayer and Merlin Dorfman, IEEE Computer Society, 1996.

[84] Joseph Goguen, Charlotte Linde, and Tora Bikson. Optimal structures for multimedia instruction. Technical report, Computer Science Lab, SRI International, July 1985.

[85] Joseph Goguen, Charlotte Linde, and Miles Murphy. Crew communication as a factor in aviation accidents. In E. James Hartzell and Sandra Hart, editors, *Papers from the 20th Annual Conference on Manual Control*. NASA Ames Research Center, 1984.

[86] Joseph Goguen and Luqi. Formal methods and social context in software development. In Peter Mosses, Mogens Nielsen, and Michael Schwartzbach, editors, *Proceedings, Sixth International Joint Conference on Theory and Practice of Software Development (TAPSOFT 95)*, pages 62–81. Springer, 1995. Lecture Notes in Computer Science, Volume 915.

[87] Joseph Goguen and Grant Malcolm. Proof of correctness of object representation. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 119–142. Prentice Hall, 1994.

[88] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.

[89] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97–538, UCSD, Dept. Computer Science & Eng., May 1997. To appear in *Theoretical Computer Science*. Early abstract in *Proc., Conf. Intelligent Systems: A Semiotic Perspective, Vol. I*, ed. J. Albus, A. Meystel and R. Quintero, Nat. Inst. Science & Technology (Gaithersberg MD, 20–23 October 1996), pages 159–167.

[90] Joseph Goguen, Grant Malcolm, and Tom Kemp. A hidden Herbrand theorem, to appear.

[91] Joseph Goguen and José Meseguer. Security policies and security models. In Marvin Schafer and Dorothy D. Denning, editors, *Proceedings, 1982 Symposium on Security and Privacy*, pages 11–22. IEEE Computer Society, 1982.

[92] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.

[93] Joseph Goguen and José Meseguer. Unwinding and inference control. In Dorothy D. Denning and Jonathan K. Millen, editors, *Proceedings, 1984 Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[94] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985.

[95] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984.

[96] Joseph Goguen and José Meseguer. Models and equality for logical programming. In Hartmut Ehrig, Giorgio Levi, Robert Kowalski, and Ugo Montanari, editors, *Proceedings, 1987 TAPSOFT*, pages 1–22. Springer, 1987. Lecture Notes in Computer Science, Volume 250.

[97] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153–162, October 1986.

[98] Joseph Goguen and José Meseguer. Software for the Rewrite Rule Machine. In Hideo Aiso and Kazuhiro Fuchi, editors, *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 628–637. Institute for New Generation Computer Technology (ICOT), 1988.

[99] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Drafts exist from as early as 1985.

[100] Joseph Goguen, José Meseguer, and David Plaisted. Programming with parameterized abstract objects in OBJ. In Domenico Ferrari, Mario Bolognani, and Joseph Goguen, editors, *Theory and Practice of Software Technology*, pages 163–193. North-Holland, 1983.

[101] Joseph Goguen, Akira Mori, and Kai Lin. Algebraic semiotics, ProofWebs and distributed cooperative proving. In Yves Bartot, editor, *Proceedings, User Interfaces for Theorem Provers*, pages 25–34. INRIA, 1997. Sophia Antipolis, 1–2 September 1997.

[102] Joseph Goguen, Akira Mori, Kai Lin, and Akiyoshi Sato. Formal tools for distributed cooperative engineering, 1998. In preparation.

[103] Joseph Goguen and Efraim Shaket. Fuzzy sets at UCLA. *Kybernetes*, 8:65–66, 1979.

[104] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison Wesley, 1985, pages 391–420.

[105] Joseph Goguen and James Thatcher. Initial algebra semantics. In *Proceedings, Fifteenth Symposium on Switching and Automata Theory*, pages 63–77. IEEE, 1974.

[106] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice Hall, 1978.

[107] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. A junction between computer science and category theory, I: Basic concepts and examples (part 1). Technical report, IBM Watson Research Center, Yorktown Heights NY, 1973. Report RC 4526.

[108] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. An introduction to categories, algebraic theories and algebras. Technical report, IBM Watson Research Center, Yorktown Heights NY, 1975. Report RC 5369.

[109] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. A junction between computer science and category theory, I: Basic concepts and examples (part 2). Technical report, IBM Watson Research Center, Yorktown Heights NY, 1976. Report RC 5908.

[110] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.

[111] Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition, 1997. Draft manuscript.

[112] Joseph Goguen and Francisco Varela. Systems and distinctions; duality and complementarity. *International Journal of General Systems*, 5:31–43, 1979.

[113] Joseph Goguen, James Weiner, and Charlotte Linde. Reasoning and natural explanation. *International Journal of Man-Machine Studies*, 19:521–559, 1983.

[114] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies*. Academic, to appear. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.

[115] Joseph Goguen and David Wolfram. On types and FOOPS. In Robert Meersman, William Kent, and Samit Khosla, editors, *Object Oriented Databases: Analysis, Design and Construction*, pages 1–22. North Holland, 1991. Proceedings, IFIP TC2 Conference, Windermere, UK, 2–6 July 1990.

[116] John Guttag. Abstract data types and the development of data structures. *Communications of the Association for Computing Machinery*, 20:297–404, June 1977.

[117] John Guttag, Ellis Horowitz, and David Musser. Abstract data types and software validation. *Communications of the Association for Computing Machinery*, 21(12):1048–1064, 1978.

[118] Christian Heath, Marina Jirotka, Paul Luff, and Jon Hindmarsh. Unpacking collaboration: the interactional organisation of trading in a city dealing room. In *European Conference on Computer Supported Cooperative Work '93*. IEEE, 1993.

[119] Martin Heidegger. The question concerning technology. In *Basic Writings*, pages 283–217. Harper and Row, 1977. Translated by David Krell; original from 1953.

[120] C.A.R. Hoare. Unification of theories: A challenge for computing science. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, pages 49–57. Springer, 1996. Lecture Notes in Computer Science, Volume 389.

[121] Shusaku Iida, Michihiro Matsumoto, Răzvan Diaconescu, Kokichi Futatsugi, and Dorel Lucanu. Concurrent object composition in CafeOBJ. Technical report, Japan Institute of Science and Technology, 1997.

[122] Theo Janssen. Compositionality. In Johan van Benthem and Alice ter Meulen, editors, *Handbook of Logic and Language*, pages 417–473. Elsevier/MIT, 1997.

[123] Marina Jirotka. Ethnomethodology and requirements engineering. Technical Report PRG-TR-92-27, Centre for Requirements and Foundations, Oxford University Computing Lab, 1991.

[124] Marina Jirotka and Joseph Goguen. *Requirements Engineering: Social and Technical Issues*. Academic, 1994.

[125] Claude Kirchner, Hélène Kirchner, and Aristide Mégrelis. OBJ for OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies*. Academic, to appear.

[126] William Labov. The transformation of experience in narrative syntax. In *Language in the Inner City*, pages 354–396. University of Pennsylvania, 1972.

[127] George Lakoff and Mark Johnson. *Metaphors we Live by*. Chicago, 1980.

[128] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[129] Saunders Mac Lane. *Mathematics: Form and Function*. Springer, 1986.

[130] Bruno Latour. *Science in Action*. Open, 1987.

[131] Bruno Latour. *Aramis, or the Love of Technology*. Harvard, 1996.

[132] F. William Lawvere. An elementary theory of the category of sets. *Proceedings, National Academy of Sciences, U.S.A.*, 52:1506–1511, 1964.

[133] Sany Leinwand, Joseph Goguen, and Timothy Winkler. Cell and ensemble architecture of the Rewrite Rule Machine. In Hideo Aiso and Kazuhiro Fuchi, editors, *Proceedings, International Conference on Fifth Generation Computer Systems 1988*, pages 869–878. Institute for New Generation Computer Technology (ICOT), 1988.

[134] Charlotte Linde and Joseph Goguen. Structure of planning discourse. *Journal of Social and Biological Structures*, 1:219–251, 1978.

[135] David Luckham, Friedrich von Henke, Bernd Krieg-Brückner, and Olaf Owe. ANNA*: A Language for Annotating Ada Programs*. Springer, 1987. Lecture Notes in Computer Science, Volume 260.

[136] Paul Luff, Marina Jirotka, Christian Heath, and David Greatbatch. Tasks and social interaction: the relevance of naturalistic analyses of conduct for requirements engineering. In Stephen Fickas and Anthony Finkelstein, editors, *Requirements Engineering '93*, pages 187–190. IEEE, 1993.

[137] Luqi and Joseph Goguen. Formal methods: Problems and promises. *IEEE Software*, 14(1):73–85, 1997.

[138] Jean-François Lyotard. *The Postmodern Condition: a Report on Knowledge*. Manchester, 1984. Theory and History of Literature, Volume 10.

[139] Grant Malcolm. Behavioural equivalence, bisimilarity, and minimal realisation. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specifications*. Springer, 1989. Lecture Notes in Computer Science, Volume 389.

[140] Grant Malcolm and Joseph Goguen. Proving correctness of refinement and implementation. Technical Report Technical Monograph PRG-114, Programming Research Group, University of Oxford, 1994. Submitted for publication.

[141] Grant Malcolm and Joseph Goguen. An executable course on the algebraic semantics of imperative programs. In Michael Hinchey and C. Neville Dean, editors, *Teaching and Learning Formal Methods*, pages 161–179. Academic, 1996.

[142] Grant Malcolm and James Worrell. Toposes of abstract machines with observational semantics, 1997. Draft, Oxford University Computing Laboratory.

[143] Humberto Maturana. Biology of language: The epistemology of reality. In George Miller and Eric Lenneberg, editors, *Psychology and Biology of Thought and Language: Essays in Honor of Eric Lenneberg*, pages 27–64. Academic, 1978.

[144] Humberto Maturana and Francisco Varela. *Autopoiesis and Cognition: The Realization of the Living*. Reidel, 1980.

[145] Humberto Maturana and Francisco Varela. *The Tree of Knowledge*. Shambhala, New Science Library, 1987.

[146] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium 1987*, pages 275–329. North-Holland, 1989.

[147] José Meseguer. Conditional rewriting logic: Deduction, models and concurrency. In Stéphane Kaplan and Misuhiro Okada, editors, *Conditional and Typed Rewriting Systems*, pages 64–91. Springer, 1991. Lecture Notes in Computer Science, Volume 516.

[148] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Aki Yonezawa, editors, *Research Directions in Object-Based Concurrency*. MIT, 1993.

[149] José Meseguer. Membership algebra as a logical framework for equational specification, 1997. Draft manuscript. Computer Science Lab, SRI International.

[150] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[151] José Meseguer and Joseph Goguen. Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. *Information and Computation*, 103(1):114–158, March 1993. Revision of a paper presented at LICS 1987.

[152] José Meseguer, Joseph Goguen, and Gert Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989.

[153] Lawrence Moss, José Meseguer, and Joseph Goguen. Final algebras, cosemicomputable algebras, and degrees of unsolvability. *Theoretical Computer Science*, 100:267–302, 1992. Original version from March 1987.

[154] Peter G. Neumann. *Computer-Related Risks*. ACM (Addison-Wesley), 1995.

[155] David Parnas. Information distribution aspects of design methodology. *Information Processing '72*, 71:339–344, 1972. Proceedings of 1972 IFIP Congress.

[156] David Parnas. A technique for software module specification. *Communications of the Association for Computing Machinery*, 15:330–336, 1972.

[157] Charles Saunders Peirce. *Collected Papers*. Harvard, 1965. In 6 volumes; see especially Volume 2: Elements of Logic.

[158] Tekla Perry. In search of the future of air traffic control. *IEEE Spectrum*, 34(8):18–35, August 1997.

[159] Wesley Phoa. Should computer scientists read Derrida? Technical Report 24/5/93, University of New South Wales, 1993. School of Computer Science and Engineering.

[160] Francisco Pinheiro and Joseph Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, pages 52–64, March 1996. Special issue of papers from ICRE'96.

[161] Horst Reichel. Initially restricting algebraic theories. In Piotr Dembinski, editor, *Mathematical Foundations of Computer Science*, pages 504–514. Springer, 1980. Lecture Notes in Computer Science, Volume 88.

[162] Harvey Sacks. An analysis of the course of a joke's telling in conversation. In Richard Baumann and Joel Scherzer, editors, *Explorations in the Ethnography of Speaking*, pages 337–353. Cambridge, 1974.

[163] Harvey Sacks. *Lectures on Conversation*. Blackwell, 1992. Edited by Gail Jefferson.

[164] Ferdinand de Saussure. *Course in General Linguistics*. Duckworth, 1976. Translated by Roy Harris.

[165] Y.V. Srinivas and Richard Jüllig. SpecWare language manual, version 2.0. Technical report, Kestrel, 1996.

[166] Victoria Stavridou, Joseph Goguen, Steven Eker, and Serge Aloneftis. FUNNEL: A CHDL with formal semantics. In *Proceedings, Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 117–144. IEEE, 1991. Turin.

[167] William Irwin Thompson, editor. *Gaia: a Way of Knowing*. Lindisfarne, 1987.

[168] William Tracz. LILEANNA: a parameterized programming language. In *Proceedings, Second International Workshop on Software Reuse*, pages 66–78, March 1993. Lucca, Italy.

[169] William Tracz. *Formal Specification of Parameterized Programs in* LILLEANNA. PhD thesis, Stanford University, 1997.

[170] Mark Turner. *The Literary Mind*. Oxford, 1997.

[171] Francisco Varela and Joseph Goguen. The arithmetic of closure. *Journal of Cybernetics*, 8:125ff, 1978. Also in *Progress in Cybernetics and Systems Research*, Volume 3, edited by R. Trappl, George Klir and L. Ricciardi, Hemisphere Publishing Co., 1978.

[172] Simone Veglioni. *Integrating Static and Dynamic Aspects in the Specification of Open, Object-based and Distributed Systems*. PhD thesis, Oxford University Computing Laboratory, 1998.

[173] Jesse Wright, James Thatcher, Eric Wagner, and Joseph Goguen. Rational algebraic theories and fixed-point solutions. In Michael J. Fischer, editor, *Proceedings, Seventeenth Symposium on Foundations of Computing*, pages 147–158. IEEE, 1976.

[174] Lotfi Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.