Software Engineering with OBJ: algebraic specification in action

Edited by Joseph Goguen and Grant Malcolm

1 Introduction

Information is the life-blood of modern society. It is largely controlled, distributed and manipulated through software systems that drive communications networks, mediate financial transactions and operate databases of almost anything from recipes, through vehicle registrations and corporate finances, to criminal records. These systems range in scope from personal organizers to networks that girdle the earth. They control access to buildings, allow researchers to communicate results and share ideas, format documents such as financial reports and this book; they monitor and control particle accelerators, production lines, nuclear reactors, satellite trajectories and ballistic missiles, as well as cash dispensers and the family car brakes. Participating in society without encountering such systems is as difficult as cutting a pound of flesh from a man's breast without shedding a jot of blood.

Information can be effectively recorded, controlled and used only insofar as it is *about* something, whether money, braking distances, mouse buttons or production quotas. How information is interpreted depends on many things, including dynamic technical, social, psychological, legal, and commercial factors. This is the *situated*, social aspect of information [19, 21] that feeds the muscles, organs, nerves and brains of society. Another important aspect of information, particularly as embodied in software systems, is that it is as precisely delineated and *structured* as a hemoglobin molecule, or the DNA that controls the body's manufacture of hemoglobin. For example, the debit and credit values in a spreadsheet are represented as sequences of zeroes and ones, and adding columns is achieved by manipulating these sequences. To design and build effective software systems, the software engineer must master both the technical and the social aspects of the relevant information [19, 37].

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. For example, the US Federal Aviation Agency (FAA) cancelled an $8 billion contract with IBM to build the next generation air traffic control system for the entire United States. This is perhaps the biggest software cancellation in history, but there are many more examples, including cancellation of a $2 billion contract with IBM to provide modern information systems to replace myriads of obsolete, incompatible systems being used by the US Department of Defense. The highly publicized failure of IBM software to deliver real time sports data to the media at the 1996 Olympic Games in Atlanta, the self-destruction of the European satellite launch Ariane 5, costing $2 billion, the 1994 failure of the United Airlines baggage delivery system at Denver International Airport, which delayed its opening by one and a half years costing about $15 million [16], and 1989 cancellation of the $2 million Taurus system, which was supposed to bring the London Stock Exchange into the 20th century and instead led to a diminution of its power. Much more information of this kind can be found in the Risks Forum run by Peter Neumann (see his column in *Communications of the ACM*, as well as `www.csl.sri.com/risks.html` and [43]).

Anyone who has worked for some time 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. For obvious reasons, the organizations involved usually try to hide such failures, but experience suggests that half or more of large complex systems fail in one way or another, and that the frightening list in the previous paragraph is just the tip of an enormous iceberg.

Experience also shows that many of these failures are due to a mismatch between the social and technical

aspects of a supposed solution, i.e., due to the tension between information as situated and as structured representation. Traditionally, software engineering has been biased towards this latter, formalistic, view of information. Understandably so, 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.

This book presents case studies in software design using the high-level specification and prototyping language OBJ. The emphasis in these studies is on the formal aspects of design, particularly the formal specification and verification of systems; however, the preceding discussion of the situatedness of information and software highlights some reasons we feel OBJ is appropriate for these endeavors. The following three sections address various aspects of this.

1.1 OBJ is a Specification Language

In the early days of computing, coding arithmetic routines involved a considerable investment of time and effort; nowadays the task is considered trivial, and building a (small) compiler is considered a suitable task for undergraduate students. The main reason behind this increased efficiency of software production, without which the pervasiveness of information systems would be impossible, is the development of high-level programming languages. These allow programmers to ignore many details and concentrate on more abstract properties of their software. For example, these languages allow a more abstract conceptualization of the data the program manipulates, e.g., numbers rather than their binary representations. Indeed, much of the history of computing can be seen as a series of advances in *abstraction* mechanisms of various kinds for various purposes, for example, from subroutines, through procedures, to modules, and now objects and repositories.

The need for abstraction is illustrated by Jonathan Swift's parody on the Academy of Lagado, where the professors communicate not with words but with the objects that they represent[1]:

> many of the most learned and wise adhere to the new scheme of expressing themselves by *things*, which hath only this inconvenience attending it, that if a man's business be very great, and of various kinds, he must be obliged in proportion to carry a greater bundle of *things* upon his back, unless he can afford one or two servants to attend him. I have often beheld two of these sages almost sinking under the weight of their packs... who when they meet in the streets would lay down their loads, open their sacks and hold conversation for an hour together; then put up their implements, help each other to resume their burthens, and take their leave.

Specification languages allow a more abstract view than programming languages. They concentrate on functionality and design structure, leaving algorithmic aspects for a later coding phase. Declarative specification languages like OBJ allow an incremental approach to developing specifications. The first stage might consist of merely naming the kinds of entity that compose the system, while later stages refine the functionalities of these entities. Algebraic specification is perhaps the best developed strain of formal specification, both in its supporting theory and in its very efficient implementation technology. As such it seems especially suitable for expressing standards, such as GKS (Chapter 5).

Of course, this is still a long way from capturing the situatedness of the end product, but we believe it is an important step in the right direction, not just because of the abstractness offered by OBJ, but also because the executability of its specifications supports prototype-driven incremental development methods.

1.2 OBJ is a Functional Programming Language

OBJ specifications consist of names for kinds of entities (sorts), and operations on those entities whose functionality is specified by equations. These equations look similar to the definitions in functional programming

[1] From *Gulliver's Travels*. Although Swift's satire is largely directed against seventeenth century ideals of conciseness in language, it is also in part directed against ideal universal languages like that proposed by Leibniz (see Section 1.5 below); we do not think he was satirizing object oriented programming.

languages (though they are more general), and they can be used to experiment with a specification. Therefore a specification can be seen as a *prototype* of its eventual implementation, so that designers, and more importantly end users, can experiment with specifications, to discover flaws and gaps in the specification or requirements statements. These can then be rectified before the expensive coding process begins. Thus OBJ can be used in an iterative prototype-driven development process, where key aspects are incrementally improved on the basis of user feedback throughout the lifecycle. Moreover, today's hardware allows many small to medium applications to be written and run directly in OBJ, without any coding at all.

1.3 OBJ is a Theorem Prover

Although its support for prototyping allows designers to take some account of the situatedness of the systems they design, OBJ is primarily intended to support the structured representation aspect of software systems. OBJ was designed for algebraic semantics: its declarations introduce symbols for sorts and functions, its statements are equations, and its computations are equational proofs. Thus, an OBJ specification actually *is* an equational theory, and every OBJ computation actually *proves* some theorem about such a theory. In the same way that equational reasoning can be used to prove properties of numbers or sets, this allows designers to prove (using OBJ itself!) properties of their specifications. Many examples appear in our book, *Algebraic Semantics of Imperative Programs* [24], along with the relevant mathematical theory, we hope in a relatively digestible form. The following chapters provide many more examples; a further example worth mentioning is the use of OBJ to specify and prove the correctness of an optimizing compiler [35]; this involves proving that one specification refines another, i.e., provides (at least) the same behavior.

1.4 OBJ has a Past and a Future

Most of the relevant historical information can be found in Section 1.1 of Chapter 1 of this book. Here we fill in some earlier and later developments, including some information about the origins of this book.

Perhaps the most important enabling event for OBJ was Goguen's gradual realization, during the period from 1968 to 1972, that Lawvere's characterization of the natural numbers as a certain initial algebra [39] could be extended to other data structures of interest to computer science; reading Knuth's compendium [38] in a seminar at the IBM T.J. Watson Research Center organized by Jim Thatcher was a great help, and the influence of Sanders Mac Lane was also important during this period.

This insight led to the development of the mathematical theory of abstract data types as initial algebras [32], as well as the somewhat earlier general theory of abstract syntax and compositional semantics [17, 31]. The attempt to develop the computational side of abstract data types led to considering term rewriting theory, and indeed an early draft of [32] made extensive use of these ideas. However, term rewriting theory was still in a primitive state at that time, and a more abstract viewpoint was found expedient. Nevertheless, this early foray into term rewriting was an essential precondition for OBJ. (For much more historical information on the so-called ADJ group, see [18].)

This book grew out of a project to implement a new version of OBJ in the UK, and use it in a number of industrial experiments. This effort was led by Derek Coleman and Robin Gallimore at UMIST (University of Manchester Institute of Science and Technology), with Victoria Stavridou as Research Assistant, supported by the UK SERC (Science and Engineering Research Council, recently renamed EPSRC); UMIST OBJ was used in several of the studies reported in this book. It was later commercialized, renamed OBJ-Ex, and supported by Gerrard Software, a small UK firm; OBJ-Ex appears to be still in use within the UK government. Meanwhile, Coleman and Gallimore moved from UMIST to Hewlett-Packard at Bristol, and developed another variant of OBJ, called Axis [8], while the UMIST project came under the direction of Colin Walter. An OBJ users group was formed in the UK, and held a number of meetings with participation by many of the authors in this book.

Several post-OBJ3 developments seem especially noteworthy. One is Maude [7, 6], an extension of OBJ to rewriting logic [41], which is particularly suited to specifying concurrent systems. This project is led by Dr. José Meseguer at SRI International in Menlo Park, California, where most of the original OBJ3 development was done, and indeed, Dr. Meseguer co-led the later phases of the OBJ3 implementation effort, while Goguen

was chief designer, and supervised further enhancements at Oxford University. As this is written, there is a preliminary release of Maude, with a very efficient implementation and several interesting features, including support for meta-programming [5] based on reflection, and for order sorted membership equational logic [42], a recent extension of order sorted equational logic.

The second system, called CafeOBJ [12, 13], has been implemented at JAIST (Japan Advanced Institute of Science and Technology) in Hokuriku, Japan, under the direction of Professor Kokichi Futatsugi, supported on a large scale by MITI, the Japanese Ministry of Industry and Technology. This system includes features to handle both rewriting logic (as in Maude) and hidden algebra [25, 26, 11], which provides powerful proof techniques for behavioral specification and verification.

A third group called "CoFI," consisting (largely) of European theoretical computer scientists, is designing and building a "common" algebraic specification language called CASL. Participants include Bernd Krieg-Bruckner, who designed the Ada module system (influenced by OBJ), Don Sannella, Peter Mosses, Till Mossakowsky, Maura Cerioli, Michel Bidoit, and Andre Tarlecki. Although it is not clear that they would like CASL to be called a descendent of OBJ, the language certainly does have significant similarities to OBJ3; perhaps its most distinctive feature is its advocacy of partial first order logic. The latest design documents can be found at `www.brics.dk/Projects/CoFI/`.

We are also pleased to report a significant event for the OBJ community, a "OBJ/CafeOBJ/Maude" workshop held at FM'99 (World Congress on Formal Methods in Toulouse, France) in September 1999, where presentations were made on many exciting new developments [15]; in addition, there was a OBJ/CafeOBJ/Maude mini-track as part of the main FM'99 event, the papers from which were published in its proceedings [47].

Finally, we should mention that part of the spirit of OBJ also lives on in the module system of several modern programming languags, including ML [36], C++, and Ada [9], since the designers of the languages were all influenced by OBJ3, and/or its predecessor Clear [4].

1.5 OBJ is not the Last Word

The case studies in this book show that OBJ can be used to specify, prototype and reason about both software and hardware systems; that it can be used to define formal semantics of languages and standards; and that it can provide support for mechanical theorem proving. Because of the clarity of its semantics, OBJ is also a useful educational tool, and it has been used at UCSD, Oxford and other universities in courses on theorem proving and on the semantics of imperative programs [40]. All this is not to suggest that we view OBJ as the *characteristica universalis* envisioned by Gottfried Leibniz in the seventeenth century:

> All our reasoning is nothing but the joining and substituting of characters, whether these characters be words or symbols or pictures... if we could find characters or signs appropriate for expressing all our thoughts as definitely and as exactly as arithmetic expresses numbers... we could in all subjects in so far as they are amenable to reasoning accomplish what is done in arithmetic and geometry. For all inquiries that depend on reasoning would be performed by the transposition of characters and by a kind of calculus.... And if someone would doubt my results, I should say to him: 'let us calculate, Sir,' and thus by taking to pen and ink, we should soon settle the question.

Leibniz was not concerned merely with a language for formal reasoning in mathematics. Anticipating by more than three hundred years the kind of predictions that earned Artificial Intelligence a reputation for hyperbole [14], Leibniz wrote:

> I believe that a number of chosen men can complete the task within five years; within two years they will exhibit the common doctrines of life, that is, metaphysics and morals, in an irrefutable calculus[2].

Nonetheless, we believe that OBJ is particularly well-suited to specifying and reasoning about the formal structures of software systems. Chapter 1 of this book gives an introduction to OBJ, describing precisely

[2] These quotations from Leibniz are taken from a fascinating discussion of mathematical formalism in Barrow [3].

how its declarations relate to equational theories, how its specifications can be executed, how it allows error specification and recovery, how its powerful module facilities support hierarchical design, and much more. The ensuing chapters show how this translates into practice.

OBJ's elegant declarative, algebraic approach makes it appropriate to many paradigms in software engineering, and it has been applied to the logic paradigm in Eqlog [28, 10], and to the object paradigm in FOOPS [29]. Chapter 1 explains how OBJ's module features support what is called *parameterized programming*, which is particularly appropriate for large-grain specification, and provides facilities for prototyping at the system design level. It also provides support for reuse of code and designs [33].

Some recent research is directed to unifying the functional, logic, constraint and object paradigms in a way that takes better account of the situatedness of software [27]. Although some elements of this paradigm are already prefigured in the use of FOOPS in TOOR [44], an object oriented environment for requirements tracing, much exciting work remains to be done. Other current research is using OBJ in the Kumo project, to verify distributed concurrent systems [22, 23], using techniques from hidden algebra [45, 30, 46, 25, 26].

The dismaying pattern of ongoing, and indeed escalating, software failure described earlier suggests that the real software crisis lies not so much in producing software as in producing software appropriate to the situation where it will actually be used. Alasdair Gray's short story 'The Crank that Turned the Revolution' [34] tells the cautionary tale of Vague McMenamy, a thoughtful and precocious child who

> would stand for long hours on the edge of the duck-pond wondering how to improve his Granny's ducks.... He thought that since ducks spend most of their days in water they should be made to do it efficiently. With the aid of a friendly carpenter he made a boat-shaped container into which a duck was inserted. There was a hole at one end through which the head stuck out, allowing the animal to breathe, see and even eat; nonetheless it protested against the confinement by struggling to get out and in doing so its wings and legs drove the cranks which conveyed motion to a paddle-wheel on each side. On its maiden voyage the duck zig-zagged around the pond at a speed of thirty knots, which was three times faster than the maximum speed which the boats and ducks of the day had yet attained. McMenamy had converted a havering all-rounder into an efficient specialist.

Encouraged by his success, McMenamy constructs a larger craft, "to be driven by every one of his Granny's seventeen ducks", powered by a screw propeller.

> Quacking hysterically, it crossed the pond with such velocity that it struck the opposite bank at the moment of departure from the near one. Had it struck soil it would have embedded itself. Unluckily, it hit the root of a tree, rebounded to the centre of the pond, overturned and sank. Every single duck was drowned.

This light-hearted illustration of the dangers of an overly formalistic view of ducks as swimming-systems should not distract us from the more serious illustrations mentioned earlier. The situatedness of information and software, and the role that this plays in system design, is an active research area. At present, most research is directed at the initial stages of requirements analysis and capture, but some recent work is directed at the relationships between the situated and formalistic aspects of software, such as the theory of situated abstract data types [19, 20, 21].

The latest information on OBJ and its progeny can be obtained over the World Wide Web, at

`http://www.cs.ucsd.edu/users/goguen/sys/obj.html`

and subscription to an OBJ mailing list is available by emailing to

`objforum@comlab.ox.ac.uk`.

1.6 Contents of this Book

We now briefly describe the papers in this volume, in some cases indicating how they relate to themes sketched earlier in this introduction.

Part I of the book consists of *Introducing OBJ*, by Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud. This is the official OBJ3 manual, with many examples and applications, including proofs as well as specifications.

Part II contains a number of OBJ specifications. Victoria Stavridou uses OBJ and REVE in the development of hardware and proofs of properties of digital systems that evolve through time. Ataru Nakagawa, Kokichi Futatsugi, S. Tomura and T. Shimizu present a specification of a sophisticated interactive graphics package, and an example of a systematic way to check specifications during the design process. David Duce applies OBJ to the GKS graphics standard, showing that executable specifications can help to find errors, and that algebraic specifications can be relatively easy to read and modify.

Part III demonstrates the specification of formal semantics in OBJ. Joseph Goguen uses OBJ to specify a new kind of parallel computer architecture based on term rewriting, called the *Rewrite Rule Machine*; this is perhaps the first formal specification of a computer architecture. The rewrite rule machine is intended to provide a platform for an efficient implementation of OBJ's executable sublanguage [1, 2]. Claude Kirchner, Hélène Kirchner and Aristide Mégrelis describe their experience building the OBJ3 interpreter using OBJ2 as a specification language; the OBJ2 code supported coding, as well as communication among a diverse team of programmers and designers. Eugenio Battison, Fiorella de Cindio and Gian-Carlo Mauri also consider parallelism in their paper, which uses OBJ to add data types and data abstraction to the Petri net approach to concurrent system specification; in particular, OBJ is used to specify the data types to which individual tokens belong.

Part IV closes the book with two papers on Parameterized Programming. Kazuhito Ohmaki, Koichi Takahashi and Kokichi Futatsugi present a LOTOS simulator in OBJ. This is used to check behavioral properties of LOTOS specifications, and includes a discussion of extending LOTOS with some of OBJ's support for modularity such as parameterized modules. Finally, Joseph Goguen and Grant Malcolm document some higher order features of OBJ's module system that were somehow omitted from Part I, and illustrate the use of these advanced features for programming and theorem proving.

Looking over this set of papers, it seems to us both surprising and impressive that OBJ can be effectively used in so many different ways for so many different purposes. Surely there is an important lesson here for the emerging field of Formal Methods: there is no single best way to use any given notation or tool; instead, great flexibility is needed to adapt to the incredible variety of projects and their social contexts. The experiences reported in these papers also reinforce our strongly held view that a formal notation or tool used for specification should be executable, and not merely printable.

References

[1] Hitoshi Aida, Joseph Goguen, Sany Leinwand, Patrick Lincoln, José Meseguer, Babak Taheri, and Timothy Winkler. Simulation and performance estimation for the Rewrite Rule Machine. In *Proceedings, Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 336–344. IEEE, October 1992.

[2] Hitoshi Aida, Joseph Goguen, and José Meseguer. Compiling concurrent rewriting onto the rewrite rule machine. In Stéphane Kaplan and Misuhiro Okada, editors, *Conditional and Typed Rewriting Systems*, pages 320–332. Springer, 1991. Lecture Notes in Computer Science, Volume 516; also, Technical Report SRI-CSL-90-03, Computer Science Lab, SRI International, February, 1990.

[3] John D. Barrow. *Pi in the Sky: counting, thinking and being.* Penguin, 1992.

[4] 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.

[5] Manuel Clavel, Francisco Duran, Steven Eker, José Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In *Proceedings, FM'99 - Formal Methods, Volume II*, pages 1684–1701. Springer, 1999. Lecture Notes in Computer Science, Volume 1709.

[6] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. 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*.

[7] Manuel Clavel, Steven Eker, and José Meseguer. Current design and implementation of the Cafe prover and Knuth-Bendix tools, 1997. Presented at CafeOBJ Workshop, Kanazawa, October 1997.

[8] Derek Coleman et al. The Axis papers. Technical report, Hewlett-Packard Labs, Bristol, September 1988.

[9] Department of Defense. Reference manual for the Ada programming language. United States Government, Report ANSI/MIL-STD-1815A, 1983.

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

[11] Răzvan Diaconescu. Behavioural coherence in object-oriented algebraic specification. Technical Report IS–RR–98–0017F, Japan Advanced Institute for Science and Technology, June 1998. Submitted for publication.

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

[13] Răzvan Diaconescu and Kokichi Futatsugi. Logical foundations of CafeOBJ, 1999. Submitted for publication.

[14] Hurbert L. Dreyfus. *What Computers Still Can't Do*. MIT, 1992.

[15] Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors. *OBJ/CafeOBJ/Maude at Formal Methods '99*. Theta (Bucharest), 1999. Proceedings of a workshop held in Toulouse, France, 20 and 22 September 1999.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] Joseph Goguen, Kai Lin, Akira Mori, Grigore Roşu, and Akiyoshi Sato. Tools for distributed cooperative design and validation. In *Proceedings, CafeOBJ Symposium*. Japan Advanced Institute for Science and Technology, 1998. Namazu, Japan, April 1998.

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

[25] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97–538, UCSD, Dept. Computer Science & Eng., May 1997. To appear in special issue of *Theoretical Computer Science* on Algebraic Engineering, edited by Chrystopher Nehaniv and Masamo Ito. Extended 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.

[26] Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 9(3):287–319, June 1999.

[27] Joseph Goguen, Grant Malcolm, and Tom Kemp. A hidden Herbrand theorem: Combining the object, logic and functional paradigms. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, pages 445–462. Springer Lecture Notes in Computer Science, Volume 1490, 1998. Full version to appear in *Electronic Journal of Functional and Logic Programming*, MIT, 1999.

[28] 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.

[29] 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.

[30] Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 – Formal Methods*, pages 1704–1719. Springer, 1999. Lecture Notes in Computer Sciences, Volume 1709, Proceedings of World Congress on Formal Methods, Toulouse, France.

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

[32] 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.

[33] Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-based Systems*. Cambridge, To appear 1999.

[34] Alasdair Gray. *Unlikely Stories, Mostly*. Penguin, 1984.

[35] Lutz Hamel. *Behavioural Verification and Implementation of an Optimizing Compiler for OBJ3*. PhD thesis, Oxford University Computing Lab, 1996.

[36] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

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

[38] Donald Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1971.

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

[40] 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.

[41] 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.

[42] José Meseguer. Membership algebra as a logical framework for equational specification. In Francisco Parisi-Presicce, editor, *Proceedings, WADT'97 – Workshop on Abstract Data Types*, pages 18–61. Springer, 1998. Lecture Notes in Computer Science, Volume 1376.

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

[44] 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.

[45] Grigore Roşu. Behavioral coinductive rewriting. In Kokichi Futatsugi, Joseph Goguen, and José Meseguer, editors, *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 179–196. Theta (Bucharest), 1999. Proceedings of a workshop held in Toulouse, France, 20 and 22 September 1999.

[46] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Ricardo Caferra and Gernot Salzer, editors, *Proceedings, 1998 Workshop on First Order Theorem Proving*, pages 213–223. Technische Universität Wien, 1998. (Schloss Wilhelminenberg, Vienna, November 23-25, 1998). Full version to appear in *Lecture Notes in Artificial Intelligence*, Springer, 1999.

[47] Jeannette Wing, Jim Woodcock, and Jim Davies, editors. *FM'99 – Formal Methods*. Springer, 1999. Lecture Notes in Computer Sciences, Volumes 1078 and 1709, Proceedings of World Congress on Formal Methods, Toulouse, France.

Contents