Hidden Algebra for Software Engineering

Joseph A. Goguen
Department of Computer Science & Engineering
University of California at San Diego, La Jolla CA 92093-0114 USA

Abstract: This paper is an introduction to recent research on hidden algebra and its application to software engineering; it is intended to be informal and friendly, but still precise. We first review classical algebraic specification for traditional "Platonic" abstract data types like integers, vectors, matrices, and lists. Software engineering also needs changeable "abstract machines," recently called "objects," that can communicate concurrently with other objects through visible "attributes" and state-changing "methods." Hidden algebra is a new development in algebraic semantics designed to handle such systems. Equational theories are used in both cases, but the notion of satisfaction for hidden algebra is *behavioral*, in the sense that equations need only *appear* to be true under all possible experiments; this extra flexibility is needed to accommodate the clever implementations that software engineers often use to conserve space and/or time. The most important results in hidden algebra are powerful *hidden coinduction* principles for proving behavioral properties. This paper also includes some comparison with the closely related area called coalgebra, and some bits of history.

1 Introduction

Algebra in its modern abstract sense seems to have begun with Emmy Noether around 1927 [76]. Today this area builds on equational theories for monoids, groups, rings, etc. Garrett Birkhoff pioneered the general study of equational theories under what we now call "loose semantics," giving in particular rules of deduction and a completeness theorem [6]; Alfred Tarski also made significant early contributions to this area, which is now called "universal algebra" (or sometimes "general algebra"). ADJ[1] pioneered "initial semantics" and its application to abstract data types (hereafter abbreviated ADTs) in computer science [26, 46, 45], which has since blossomed into an area of research called "algebraic semantics" (also "algebraic specification"), with hundreds of workers, its own conferences and journals, and even its own professional society (called AMAST). Hidden algebra is one of the most recent developments in this lively area.

1.1 Notes on the State of the Software Arts

Software development is very difficult. To understand it better, we can distinguish among designing, coding, and verifying (i.e., proving properties of) programs. Most of the literature addresses code verification, but this is very difficult in practice, and empirical studies have shown that little of the cost of software arises from errors in coding: most comes from errors in design and requirements [7]. Moreover, many programs are written in obscure and/or obsolete languages, with complex ugly semantics (like Cobol, Jovial, and Mumps), are very poorly documented, are indispensable to some enterprise, and are very large, often several million lines, sometimes more. Therefore it is usually an enormous effort to verify real code, and it isn't usually worth the trouble. I like to call this the *semantic swamp*; it is a place to avoid.

Moreover, programs in everyday use usually evolve, because computers, operating systems, tax laws, user requirements, etc. are all changing rapidly. Therefore the effort of verifying yesterday's version is wasted, because even small code modifications can require large proof modifications – proof is a discontinuous function of truth.

[1] This name was used for the set {Goguen, Thatcher, Wagner, Wright}; see [29] for historical details.

This suggests that we should focus on *design* and *specification*. But even this is difficult, because the properties that people really want, such as security, deadlock freedom, liveness, ease of use, and ease of maintenance, are complex, not always formalizable, and even when they are formalizable, may involve subtle interactions among remote parts of systems. However, this is an area where mathematics can make a contribution.

It is well known that most of the effort in programming goes into debugging and maintaining (i.e., improving and updating) programs [7]. Therefore anything that can be done to ease these processes has enormous economic potential. One step in this direction is to "encapsulate data representations"; this means to make the actual structure of data invisible, and to provide access to it only via a given set of operations which retrieve and modify the hidden data structure. Then the implementing code can be changed without having any effect on other code that uses it. On the other hand, if client code relies on properties of the representation, it can be extremely hard to track down all the consequences of modifying a given data structure (say, changing a doubly linked list to an array), because the client code may be scattered all over the program, without any clear identifying marks. This is why the so-called year 2,000 (Y2K) problem is so difficult.

An encapsulated data structure with its accompanying operations is called an *abstract data type*. The crucial advance was to recognize that operations should be associated with data representations; this is exactly the same insight that advanced algebra from mere *sets* to *algebras*, which are sets *with* their associated operations. In software engineering this insight seems to have been due to David Parnas [71, 70], and in algebra to Emmy Noether [76]. (Parallel developments in software engineering and abstract algebra are a theme of this paper.)

It turns out that although abstraction as isomorphism is enough for algebras representing data values (numbers, vectors, etc.), other important problems in software engineering need the more general notion of *behavioral abstraction*, where two models are considered abstractly the same if they exhibit the same behavior. The usual many sorted algebra is not rich enough for this: we have to add structure to distinguish sorts used for data values from sorts used for states, and we need a more general, behavioral, notion of satisfaction, as discussed in Section 3.

In line with the above general discussion of software methodology, we want to prove properties of specifications, not properties of code. Often the most important property of a specification is that it *refines* another specification, in the sense that any model (i.e., any code realizing) the second is also a model of the first. Methodologically speaking, a refinement embodies a set of closely related design decisions for realizing one set of behaviors from another[2]. In line with the discussion of the previous paragraph, we will often want to prove *behavioral* refinements. Behavioral refinement is much more general than ordinary refinement, and many of the clever implementation techniques that so often occur in practice require this extra generality.

The need for improved software development methods is very great. Large complex software systems fail much more often than seems to be generally recognized. One highly visible example is the 1996 cancellation by the US Federal Aviation Agency of an 8 billion dollar contract with IBM to build the next generation air traffic control system for the entire US [72]. This is perhaps the largest default in history, but there are many more examples, including the 1995 cancellation by the US Department of Defense of a 2 billion dollar contract with IBM to provide modern information systems to replace myriads of obsolete, incompatible systems. Other highly publicized failures include IBM software for delivering real time sports data to the media at the 1996 Olympic Games in Atlanta, the 2 billion dollar loss of the European Ariane 5 satellite, and the failure of the software for the

[2]Empirical studies show that real software development projects involve many false starts, redesigns, prototypes, patches, etc. [11]. Although an idealized view of a project as a sequence of refinements is a useful way to organize and document work retrospectively, it is important not to confuse this with the actual development process.

United Airlines baggage delivery system at Denver International Airport, delaying its 1994 opening by one and a half years [25], and losing 1.1 million dollars per day.

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 frequently with loud declarations 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. Today attention has shifted to the Y2K problem. The cost of necessary software repairs has been estimated to lie in the multi-trillion dollar range. It remains to be seen what the effects will be when that fateful night arrives.

1.2 Overview of this Paper

Section 2 briefly summarizes some basics of classical algebraic specification theory, following [30] and [32]. Then Section 3 explains why software engineering also needs objects, and why their behavioral properties are important, followed by a brief overview of hidden algebra that follows [38]. The notation of OBJ3 [37, 48] is used in some examples. Section 4 briefly discusses some other related work.

Although I'm far from the only one working on these kinds of problem, this survey is focused on the work with which I'm most familiar, from UCSD and from the Japanese CafeOBJ project. The UCSD group has put much material on the web, including several hidden algebraic proofs with tutorial background information, remote proof execution, and Java applets to illustrate the main ideas; see `http://www.cs.ucsd.edu/groups/tatami` and the papers [35, 34, 43], which are available (along with many others) from my website, `http://www.cs.ucsd.edu/users/goguen`.

2 Algebraic Specification of "Platonic" Abstract Data Types

In the early 1970s, the ADT concept was understood due to the pioneering work of David Parnas [70, 71], but there was no precise semantics, so it was impossible to verify the correctness of an implementation for an ADT, or even to formulate what correctness might mean. The first to try an algebraic solution to this problem was Steve Zilles [77], or at least, hearing about his work via Jack Dennis led to my giving it a try. The initial algebra theory of abstract data types provided the first rigorous formalization of ADTs; this approach works especially well for things like integers, which do not change over time[3]; it also works for ADTs with state, but not as well. John Guttag [49] developed a different approach, which while not quite rigorous, was actually more suitable for handling states. This section gives precise definitions for the initial algebra approach to ADTs, and some of their basic properties, especially that an ADT is uniquely determined by its specification as an initial algebra, that abstract data types are indeed abstract, and that initial algebra semantics can capture all computable data types.

2.1 Signature, Algebra and Homomorphism

We begin with syntax, which raises more issues than you might suspect at first. The classical one sorted case of Birkhoff, Tarski etc. is not adequate for computer science applications, which involve

[3]These are the sort of eternal "ideas" with which Plato was concerned, as opposed to the later view of Aristotle, which was more concerned with change.

many different "types" of data. Benabou [1] seems to have been the first to consider many sorted universal algebra, which he did in an elegant category theoretic setting; it has since been developed by many others in more conventional settings; see [40] for more historical information. Goguen [26] introduced *overloaded* operation symbols for many sorted algebra. This extension is important for applications like dynamic binding in object oriented programming, and it is now quite common in the computer science literature. The approach[4] is based on many sorted sets: Given a set S, whose elements are called **sorts**, then an S-**sorted set** A is a family of sets A_s, one for each $s \in S$; the notation $A = \{A_s \mid s \in S\}$ will be used for such sets. Recall that S^* denotes the set of strings over S; we will let "[]" denote the empty string. We are now ready for the main concepts.

Definition 2.1 An S-sorted **signature** Σ is an $(S^* \times S)$-sorted set $\{\Sigma_{w,s} \mid \langle w, s\rangle \in S^* \times S\}$. The elements of $\Sigma_{w,s}$ are called **operation symbols** of **arity** w, **sort** s, and **rank** $\langle w, s\rangle$; in particular, $\sigma \in \Sigma_{[],s}$ is a **constant symbol**. Σ is a **ground signature** iff $\Sigma_{[],s} \cap \Sigma_{[],s'} = \emptyset$ whenever $s \neq s'$ and $\Sigma_{w,s} = \emptyset$ unless $w = []$. $\square$

By convention, $|\Sigma| = \bigcup_{w,s} \Sigma_{w,s}$ and $\Sigma' \subseteq \Sigma$ means $\Sigma'_{w,s} \subseteq \Sigma_{w,s}$ for each w, s. Similarly, **union** is defined by $(\Sigma \cup \Sigma')_{w,s} = \Sigma_{w,s} \cup \Sigma'_{w,s}$. A common special case is union with a ground signature X, for which we use the notation $\Sigma(X) = \Sigma \cup X$.

Definition 2.2 A Σ-**algebra** A consists of an S-sorted set also denoted A, plus an **interpretation** of Σ in A, which is a family of arrows $i_{s_1...s_n,s} \colon \Sigma_{s_1...s_n,s} \to [A^{s_1...s_n} \to A_s]$ for each rank $\langle s_1...s_n,\ s\rangle \in S^* \times S$, which interpret the operation symbols in Σ as actual operations on A. For constant symbols, the interpretation is given by $i_{[],s} \colon \Sigma_{[],s} \to A_s$. Usually we write just σ for $i_{w,s}(\sigma)$, but if we need to make the dependence on A explicit, we may write σ_A. A_s is called the **carrier** of A of sort s.

Given Σ-algebras A, A', a Σ-**homomorphism** $h \colon A \to A'$ is an S-sorted arrow $h \colon A \to A'$ such that $h_s(\sigma_A(a_1, ..., a_n)) = \sigma_{A'}(h_{s_1}(s_1), ..., h_{s_n}(a_n))$ for each $\sigma \in \Sigma_{s_1...s_n,s}$ and $a_i \in A_{s_i}$ for $i = 1, ..., n$, and such that $h_s(c_A) = c_{A'}$ for each constant symbol $c \in \Sigma_{[],s}$. $\square$

An increased emphasis on homomorphisms characterizes the modern era of general algebra; this is related to the central role that morphisms play in category theory. Given homomorphisms $f \colon A \to B$ and $g \colon B \to C$, we denote their composition $A \to C$ by $f; g$. Also we denote the identity homomorphism on an algebra A by 1_A.

2.2 Term, Equation and Specification

Given an S-sorted signature Σ, the S-sorted set T_Σ of (**ground**) Σ-**terms** is the smallest set of lists of symbols that contains the constants (i.e., $\Sigma_{[],s} \subseteq T_{\Sigma,s}$), and such that given $\sigma \in \Sigma_{s_1...s_n,s}$ and $t_i \in T_{\Sigma,s_i}$ then $\sigma(t_1 \ldots t_n) \in T_{\Sigma,s}$. We view T_Σ as a Σ-algebra by interpreting $\sigma \in \Sigma_{[],s}$ as just σ, and $\sigma \in \Sigma_{s_1...s_n,s}$ as the operation sending $t_1, \ldots, t_n$ to the list $\sigma(t_1 \ldots t_n)$. Then T_Σ is called the Σ-**term algebra**. Note that because of overloading, terms do not always have a unique parse. The following is the key property of this algebra:

Theorem 2.3 (Initiality) Given a signature Σ with no overloaded constants[5] and a Σ-algebra M, there is a unique Σ-homomorphism $T_\Sigma \to M$. $\square$

[4] This was developed for my course Information Science 329, Algebraic Foundations of Computer Science, first taught in 1969 at the University of Chicago.

[5] Actually, every signature Σ has an initial Σ-algebra, but when Σ has overloaded constants, terms must be annotated by their sort; we will use the same notation T_Σ for this case.

This is proved in many places, for example [45]. The Σ-term algebra T_Σ serves as a standard model for a specification with no equations. For example, if Σ is the signature for the natural numbers with just zero and successor, then T_Σ is the natural numbers in Peano notation. If Σ consists of the operation symbols `0`, `s`, `+` and `*`, then T_Σ consists of all *expressions* formed using these symbols (with the right arities); these are simple numerical expressions. In order to get the natural numbers with addition and multiplication from this, we need to impose some equations.

Definition 2.4 A Σ-**equation** consists of a ground signature X of **variable symbols** (disjoint from Σ) plus two $\Sigma(X)$-terms of the same sort $s \in S$; we may write such an equation abstractly in the form $(\forall X)\ t = t'$ and concretely in the form $(\forall x, y, z)\ t = t'$ when $|X| = \{x, y, z\}$ and the sorts of x, y, z can be inferred from their uses in t and in t'. A **specification** is a pair (Σ, E), consisting of a signature Σ and a set E of Σ-equations. □

Conditional equations can be defined in a similar way, but we omit this here. Given Σ and a ground signature X disjoint from Σ, we can form the $\Sigma(X)$-algebra $T_{\Sigma(X)}$ and then view it as a Σ-algebra by forgetting the names of the new constants in X; let us denote this Σ-algebra by $T_\Sigma(X)$. It has the following universal **freeness** property:

Proposition 2.5 Given a Σ-algebra A and an interpretation $a\colon X \to A$, there is a unique Σ-homomorphism $\overline{a}\colon T_\Sigma(X) \to A$ extending a, in the sense that $\overline{a}_s(x) = a_s(x)$ for each $x \in X_s$ and $s \in S$. □

Definition 2.6 A Σ-algebra A **satisfies** a Σ-equation $(\forall X)\ t = t'$ iff for any $a\colon X \to A$ we have $\overline{a}(t) = \overline{a}(t')$ in A, written $A \models_\Sigma (\forall X)\ t = t'$. A Σ-algebra A satisfies a set E of Σ-equations iff it satisfies each one, written $A \models_\Sigma E$. We may also say that A is a P-algebra, and write $A \models P$ where $P = (\Sigma, E)$. The class of all algebras that satisfy P is called the **variety** defined by P. Given sets E and E' of Σ-equations, let $E \models E'$ mean $A \models E'$ for all E-models A. □

The following simple result is much used in equational theorem proving, but is rarely stated explicitly. Its proof is very simple because it uses the *semantics* of satisfaction rather than some particular rules of deduction, and because it exploits the *initiality* of the term algebra. We have found this typical of proofs in this area; commutative diagrams and other universal properties also help give elegant conceptual proofs.

Fact 2.7 (Lemma of Constants) Given a signature Σ, a ground signature X disjoint from Σ, a set E of Σ-equations, and $t, t' \in T_{\Sigma(X)}$, then $E \models_\Sigma (\forall X)\ t = t'$ iff $E \models_{\Sigma \cup X} (\forall \emptyset)\ t = t'$.

Proof: Each condition is equivalent to the condition that $\overline{a}(t) = \overline{a}(t')$ for every $\Sigma(X)$-algebra A satisfying E and every $a\colon X \to A$. □

Theorem 2.8 $T_{\Sigma,E} = T_\Sigma/{\equiv_E}$ is an initial (Σ, E)-algebra, where $\equiv_E$ is the Σ-congruence relation generated by the ground instances of equations in E. □

The proof can be found in many places, e.g., [45].

Usually we want proofs about software to be independent of how the data types involved happen to be represented; for example, we are usually not interested in properties of the decimal or binary representations of the natural numbers, but instead are interested in abstract properties of the abstract natural numbers. The following result shows that satisfaction of an equation by an algebra is an "abstract" property, in the sense that it is independent of how the algebra happens to be represented. This result implies that exactly the same equations are true of any one initial P-algebra as any other.

Proposition 2.9 Given a specification $P = (\Sigma, E)$, any two initial P-algebras are Σ-isomorphic; in fact, if A and A' are two initial P-algebras, then the unique Σ-homomorphisms $A \to A'$ and $A' \to A$ are both isomorphisms, and indeed, are inverse to each other. Moreover, given isomorphic Σ-algebras A and A', and given a Σ-equation e, then $A \models e$ iff $A' \models e$.

Proof: Let $f\colon A \to A'$ and $g\colon A' \to A$ be the two homomorphisms guaranteed by initiality. Now notice that the unique homomorphisms $A \to A$ and $A' \to A'$ must each be identities, namely 1_A and $1_{A'}$. Therefore $f;g\colon A \to A$ is 1_A, and $g;f\colon A' \to A'$ is $1_{A'}$. The proof of the second assertion is omitted here (see [30]). □

The word "abstract" in the phrase "abstract algebra" means "uniquely defined up to isomorphism"; for example, an "abstract group" is an isomorphism class of groups, indicating that we are not interested in properties of any particular representation, but only in properties that hold for all representations; e.g., see [57]. Because Theorem 2.9 implies that all the initial models of a specification $P = (\Sigma, E)$ are abstractly the same in precisely this sense, the word "abstract" in "abstract data type" has *exactly* the same meaning. This is not a mere pun, but a significant fact about software engineering.

Another fact suggesting we are on the right track is that any computable abstract data type has an equational specification; moreover, this specification tends to be reasonably simple and intuitive in practice. The following result from [65] slightly generalizes the original version due to Bergstra and Tucker [2]. (M is **reachable** iff the unique Σ-homomorphism $T_\Sigma \to M$ is surjective):

Theorem 2.10 (Adequacy of Initiality) Given any computable reachable Σ-algebra M with Σ finite, there is a finite specification $P = (\Sigma', A')$ such that $\Sigma \subseteq \Sigma'$, such that Σ' has the same sorts as Σ, and such that M is Σ-isomorphic to T_P viewed as a Σ-algebra. □

We do not here define the concept of a "computable algebra", but it corresponds to what one would intuitively expect: all carrier sets are decidable and all operations are total computable functions; see [65]. This result tells us that all of the data types that are of interest in computer science can be defined using initiality, although sometimes it may be necessary to add some auxiliary functions. All of this motivates the following fundamental conceptualization, which goes back to 1975 [46, 45]:

Definition 2.11 The **abstract data type** (abbreviated **ADT**) defined by a specification P is the class of all initial P-algebras. □

The importance of initiality for computer science developed gradually. The term "initial algebra semantics" and its first applications appeared in [26]; these included formulating (Knuthian) attribute semantics as a homomorphism from an initial many sorted syntactic algebra generated by a context free grammar, into a semantic algebra. In his PhD thesis [58], William Lawvere used the characterization of the natural numbers as an initial algebra in the axiom of infinity for his category theoretical axiomatization of sets. A key step in formalizing ADTs this way was my realization that Lawvere's use of initiality could be extended to characterize other data types abstractly; see [46], and for a more complete and rigorous exposition, see [45]. More on initiality can be found in [47] and [65]; the latter especially develops connections with induction and computability. See [29] for more historical information about this period, and [37, 32] for more recent results, examples and references. The adequacy of initiality was studied by Bergstra and Tucker in an important series of papers, of which [2] is most relevant for present purposes.

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. Order sorted algebra (abbreviated **OSA**) is an attempt to extend many sorted algebra to address these problems. Introduced in [28], this reached fruition in joint work with Meseguer [41, 66]; OSA provides a partial ordering relation on sorts, interpreted semantically as subset inclusion among model carriers. Meseguer and I proved [66] that there are simple ADTs with no adequate many sorted equational specification, because (what we call) the constructor-selector problem can't be solved in this setting. Order sorted initial algebras capture all *partial* recursive functions (and algebras), not just the total ones, as in the many sorted case. OSA is only slightly more difficult than many sorted algebra, and essentially all results generalize without much fuss; in particular, there are complete rules of deduction and initial models. Because OSA is strongly typed, many terms that intuitively should be well formed because they evaluate to well formed terms, are actually ill formed; [41] introduced *retracts* to handle this problem. There are now many different variants and extensions of OSA, too numerous to mention here, although Meseguer's membership equational logic [64] should not be omitted.

2.3 OBJ

Because it is all too easy to write incorrect specifications, it is important to have a tool that can not only check syntax but also execute test cases, and verify properties; such a tool is also very helpful in teaching. Around 1974, I conceived the OBJ language for this purpose, using order sorted algebra[6] with overloaded mixfix syntax, and with term rewriting as its operational semantics [27]; the goal was to make specifications as readable and testable as possible. The final OBJ3 [48] version[7] of OBJ resulted from the efforts of many people, including José Meseguer, Kokichi Futatsugi, Jean-Pierre Jouannaud, Claude and Hélène Kirchner, David Plaisted, and Joseph Tardo [44, 42, 22, 48]; this system provided loose and initial semantics, rewriting modulo equations, generic modules, order sorted algebra with retracts, and user definable builtins[8]; OBJ2 was heavily used in designing OBJ3 [56], 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 [21], which was used in the LOTOS hardware description language; the CafeOBJ system [23, 20], an industrial strength version of the OBJ language being built by a large Japanese national project, that includes hidden algebra to handle behavioral properties of objects with states; the CASL system [15], being developed by a diverse European group called CoFI (for Common Framework Initiative), which aspires to be a Common Algebraic Specification Language, although it does not support order sorted algebra or hidden algebra; and Maude [13], which efficiently extends OBJ3 with rewriting logic [64], membership logic [64], and reflection.

OBJ has notation for expressing both initial and loose specifications, using overloaded order sorted syntax [48, 37, 32]. OBJ modules that are to be interpreted loosely begin with the keyword `theory` (or `th`) and close with the keyword `endth`. Between these two keywords come declarations for sorts and operations, plus (as discussed later) variables and equations. For example, the following OBJ3 code specifies the theory of automata:

```
th AUTOM is
  sorts Input State Output .
  op s0 : -> State .
  op f : Input State -> State .
  op g : State -> Output .
endth
```

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

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

[8] 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 [48].

Any number of sorts can be declared following `sorts` (or equivalently, `sort`), and operations are declared with their arity between the : and the `->`, with their sort following the `->`.

The keyword pair `obj...endo` indicates that initial semantics is intended. For example, the Peano natural numbers are given by

```
obj NATP is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat .
endo
```

which uses "mixfix" syntax for the successor operation symbol: in the expression before the colon, the underbar character is a place holder, showing where the operation's arguments should go; hence successor has prefix syntax here.

The following implements a simple flag object by representing its state as a natural number. The keyword `pr` indicates an importation, in this case of `EVENAT`, which is the natural numbers enriched with a Boolean valued function `even`, which is true iff its argument is even; the expression after `EVENAT` renames the sort `Nat` of `EVENAT` to be `Flag`. The next line introduces three prefix operations all at once, each with the same source and target; these are methods for changing state. The line that begins with `var` declares a variable and its sort, and the next three lines are equations that define the three methods and the Boolean valued attribute `up?`.

```
obj NFLAG is
  pr EVENAT *(sort Nat to Flag) .
  ops (up_)(dn_)(rev_) : Flag -> Flag .
  op up?_ : Flag -> Bool .
  var F : Flag .
  eq up F = 2 * F .
  eq dn F = 2 * F + 1 .
  eq rev F = F + 1 .
  eq up? F = even F .
endo
```

The next section gives an abstract specification for this object.

All the OBJ3 code in this paper is executable, and (once suitable definitions for `EVENAT` and one other module are added) executing it actually proves the simple result about flags discussed below; the OBJ output is given in Appendix A.

3 Hidden Algebra

Initial semantics works very well for data structures like integers, lists, booleans, vectors and matrices, but is more awkward for situations that involve a state, i.e., an internal representation that is changed by commands and never viewed directly, but only through external "attributes." For example, it is usually more appropriate to view stacks as machines with an encapsulated (invisible) internal state, having "top" as an attribute. Although initial models exist for any reasonable specification of stacks, real stacks are more likely to be implemented by a model that is not initial, such as a pointer plus an array. This implies that a new notion of implementation is needed, different from the simple notion of initial model. Moreover, in considering (for example) stacks of integers, the sorts for stacks and for integers must be treated differently, since the latter are still modeled initially as data. Although these issues have been successfully addressed in an initial framework (e.g., [45]), it is really better to take a different viewpoint.

Hidden algebra explicitly distinguishes between "visible" sorts for data and "hidden" sorts for states. It makes sense to declare a fixed collection of shared data values, bundled together in a single algebra, because the components of a system must use the same representations for the data that they share, or else they cannot communicate[9].

Definition 3.1 Let D be a fixed **data algebra**, with Ψ its signature and V its sort set, such that each D_v with $v \in V$ is non-empty and for each $d \in D_v$ there is some $\psi \in \Psi_{[],v}$ such that ψ is interpreted as d in D; we call V the **visible sorts**. For convenience, we assume $D_v \subseteq \Psi_{[],v}$ for each $v \in V$. $\square$

The above concerns semantics; but the prudent verifier needs an effective specification for data values to support proofs, and it is especially convenient to use initial algebra semantics for this purpose, because it supports proofs by induction. We now generalize the notion of signature:

Definition 3.2 A **hidden signature (over a data algebra** (V, Ψ, D)**)** is a pair (H, Σ), where H is a set of **hidden sorts** disjoint from V, Σ is an $S = (H \cup V)$-sorted signature with $\Psi \subseteq \Sigma$, such that

(S1) each $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$ lies in $\Psi_{w,s}$, and
(S2) for each $\sigma \in \Sigma_{w,s}$ at most one hidden sort occurs in w.

We may abbreviate (H, Σ) to just Σ. If $w \in S^*$ contains a hidden sort, then $\sigma \in \Sigma_{w,s}$ is called a **method** if $s \in H$, and an **attribute** if $s \in V$. If $w \in V^*$ and $s \in H$, then $\sigma \in \Sigma_{w,s}$ is called a (**generalized**) **hidden constant**.

A **hidden** (or **behavioral**) **theory** (or **specification**) is a triple (H, Σ, E), where (H, Σ) is a hidden signature and E is a set of Σ-equations that does not include any Ψ-equations; we may write (Σ, E) or just E for short. $\square$

Condition (S1) expresses data encapsulation, that Σ cannot add any new operations on data items. Condition (S2) says that methods and attributes act singly on (the states of) objects. Every operation in a hidden signature is either a method, an attribute, or else a constant[10]. Equations about data (Ψ-equations) are not allowed in specifications; any such equation needed as a lemma should be proved and asserted separately, rather than being included in a specification. The following example should help to clarify this definition; it is the simplest possible example where something beyond pure equational reasoning and induction is needed.

Example 3.3 Below is a hidden specification for flag objects, where intuitively a flag can be either up or down, with methods to put it up, to put it down, and to reverse it:

```
th FLAG is sort Flag .
   pr DATA .
   ops (up_) (dn_) (rev_) : Flag -> Flag .
   op up?_ : Flag -> Bool .
   var F : Flag .
```

[9]In practice, there may be multiple representations for data with translations among them, and representations may change during development, so this is a simplifying assumption; however, it can easily be relaxed.

[10]In our most recent work [75], this assumption is weakened by allowing operations with more than one hidden sort; this is important for examples like sets, where union and intersection have two hidden arguments.

```
    eq up? up F = true .
    eq up? dn F = false .
    eq up? rev F = not up? F .
  endth
```

Here FLAG is the name of the module and Flag is the name of the class of flag objects. The methods and attribute are as in the previous NFLAG example, which is actually a model of the above specification, in a sense made precise below. □

If Σ is the signature of FLAG, then Ψ is a subsignature of Σ, and so a model of FLAG should be a Σ-algebra whose restriction to Ψ is D, providing functions for all the methods and attributes in Σ, and behaving as if it satisfies the given equations. Elements of such models are possible states for Flag objects. This motivates the following:

Definition 3.4 Given a hidden signature (H, Σ), a **hidden Σ-algebra** A is a (many sorted) Σ-algebra A such that $A\restriction_\Psi = D$. □

We next define behavioral satisfaction of an equation, an idea introduced by Reichel [73]. Intuitively, the two terms of an equation 'look the same' under every 'experiment' consisting of some methods followed by an 'observation,' i.e., an attribute. More formally, such an experiment is given by a *context*, which is a term of visible sort having one free variable of hidden sort:

Definition 3.5 Given a hidden signature (H, Σ) and a hidden sort h, then a Σ-**context** of sort h is a visible sorted Σ-term having a single occurrence of a new variable symbol z of sort h. A context is **appropriate** for a term t iff the sort of t matches that of z. Write $c[t]$ for the result of substituting t for z in the context c.

A hidden Σ-algebra A **behaviorally satisfies** a Σ-equation $(\forall X)\ t = t'$ iff for each appropriate Σ-context c, A satisfies the equation $(\forall X)\ c[t] = c[t']$; then we write $A \models\!\!\equiv_\Sigma (\forall X)\ t = t'$.

A **model** of a hidden theory $P = (H, \Sigma, E)$ is a hidden Σ-algebra A that behaviorally satisfies each equation in E. Such a model is also called a (Σ, E)-**algebra**, or a P-algebra, and then we write $A \models\!\!\equiv P$ or $A \models\!\!\equiv_\Sigma E$. Also we write $E' \models\!\!\equiv_\Sigma E$ iff $A \models\!\!\equiv_\Sigma E'$ implies $A \models\!\!\equiv_\Sigma E$ for each hidden Σ-algebra A. □

Example 3.6 Let's look at a simple Boolean cell C as a hidden algebra. Here, $C_{\texttt{Flag}} = C_{\texttt{Bool}} = \{true, false\}$, up $F = true$, dn $F = false$, up? $F = F$, and rev $F = not\ F$.

A more complex implementation H keeps complete histories of interactions, so that the action of a method is merely to concatenate its name to the front of a list of method names. Then $H_{\texttt{Flag}} = \{up, dn, rev\}^*$, the lists over $\{up, dn, rev\}$, while $H_{\texttt{Bool}} = \{true, false\}$, up $F = up ^\frown F$, dn $F = dn ^\frown F$, rev $F = rev ^\frown F$, while up? $up ^\frown F = true$, up? $dn ^\frown F = false$, and up? $rev ^\frown F = not$ up? F, where $^\frown$ is the concatenation operation. Note that C and H are *not* isomorphic. □

For visible equations, there is no difference between ordinary satisfaction and behavioral satisfaction. But these concepts can be very different for hidden equations. For example,

```
 rev rev F = F
```

is strictly satisfied by the Boolean cell model C, but it is *not* satisfied by the history model H, nor by the model NFLAG, for which the left side has value F+2. However, the equation is *behaviorally* satisfied by all these models. This illustrates why behavioral satisfaction is so often more appropriate for computer science applications.

Previously we gave a semantic definition of an abstract data type as an isomorphism class of initial algebras for some specification; equivalently, by Theorem 2.10, we could define it to be an isomorphism class of computable algebras, or in the order sorted case, of partial computable algebras. The hidden analog of this defines an **abstract object** (or **machine**) to be a class of all hidden algebras that satisfy some hidden specification (in practice, it is often desirable to restrict attention to reachable models).

3.1 Coinduction

The first effective algebraic proof technique for behavioral properties was context induction, introduced by Rolf Hennicker [51] and developed further in joint work with Michel Bidoit [5, 4]. Their research programme is similar to ours in several ways, but is more concerned with semantics than with proofs. Unfortunately, context induction can be awkward to apply in practice, as first noticed in [24]. We proposed hidden coinduction as a way to avoid this awkwardness. The technique resembles one introduced by Robin Milner [67], but is more general. Peter Padawitz is also developing similar notions [68, 69], using a very general notion of sentence.

Induction is a standard technique for proving properties of initial (or more generally, reachable) algebras of a theory, and principles of induction can be justified from the fact that an initial algebra has no proper subalgebras [32, 65]. Final (terminal) algebras play an analogous role[11] in justifying reasoning about behavioral properties with hidden coinduction. Before describing the final algebra, note that its use is not precisely dual to that of the initial algebra for abstract data types. The semantics of a hidden specification is not the final algebra, but rather is the variety of all hidden algebras that satisfy the spec; in fact, final algebras do not even exist in general. However, their existence for certain signatures, with no equations, plays an important technical role.

Given a hidden signature Σ without generalized hidden constants (recall these are hidden operations with no hidden arguments), the hidden carriers of the final Σ-algebra F_Σ are given by the following "magical formula," for h a hidden sort:

$$F_{\Sigma,h} = \prod_{v \in V} [C_\Sigma[z_h]_v \to D_v] \ ,$$

the product of the sets of functions taking contexts to data values (of appropriate sort). Elements of F_Σ can be thought of as 'abstract states' represented as functions on contexts, returning the data values resulting from evaluating a state in a context. This also appears in the way F_Σ interprets attributes: let $\sigma \in \Sigma_{hw,v}$ be an attribute, let $p \in F_{\Sigma,h}$ and let $d \in D_w$; then we define $F_{\Sigma,\sigma}(p,d) = p_v(\sigma(z_h,d))$; i.e., p_v is a function taking contexts in $C_\Sigma[z_h]_v$ to data values in D_v, so applying it to the context $\sigma(z_h,d)$ gives the data value resulting from that experiment. Methods are interpreted similarly; see [38] for details.

Definition 3.7 Given a hidden signature Σ, a hidden subsignature $\Phi \subseteq \Sigma$, and a hidden Σ-algebra A, then **behavioral Φ-equivalence** on A, denoted $\equiv_\Phi$, is defined as follows, for $a, a' \in A_s$:

(E1) $a \equiv_{\Phi,s} a'$ iff $a = a'$

when $s \in V$, and

(E2) $a \equiv_{\Phi,s} a'$ iff $A_c(a) = A_c(a')$ for all $v \in V$ and all $c \in C_\Phi[z]_v$

when $s \in H$, where z is of sort s and A_c denotes the function interpreting the context c as an operation on A, that is, $A_c(a) = \theta_a^*(c)$, where θ_a is defined by $\theta_a(z) = a$ and θ_a^* denotes the free extension of θ_a.

When $\Phi = \Sigma$, we may call $\equiv_\Phi$ just **behavioral equivalence** and denote it $\equiv$.

[11] Though no longer so in our latest work [75].

For $\Phi \subseteq \Sigma$, a **hidden Φ-congruence** on a hidden Σ-algebra A is a Φ-congruence $\simeq$ which is the identity on visible sorts, i.e., such that $a \simeq_v a'$ iff $a = a'$ for all $v \in V$ and $a, a' \in A_v = D_v$. We call a hidden Σ-congruence just a **hidden congruence**. □

The key property is the following:

Theorem 3.8 If Σ is a hidden signature, Φ is a hidden subsignature of Σ, and A is a hidden Σ-algebra, then behavioral Φ-equivalence is the *largest* behavioral Φ-congruence on A. □

There is beautiful abstract proof of this result for those who know a little category theory in [38]; it uses the existence of final algebras when there are no hidden constants, which is also shown in [38]. The congruence relation can be seen as a generalization of the so called Nerode equivalence in the classical theory of abstract machines, e.g., see [65].

Theorem 3.8 implies that if $a \simeq a'$ under some hidden congruence $\simeq$, then a and a' are behaviorally equivalent. This justifies a variety of techniques for proving behavioral equivalence (see also [36]). In this context, a relation may be called a **candidate relation** before it is proved to be a hidden congruence. Probably the most common case is $\Phi = \Sigma$, but the generalization to smaller Φ is useful, for example in verifying behavioral refinements.

Example 3.9 Let A be any model of the `FLAG` theory in Example 3.3, and for $f, f' \in A_{\texttt{Flag}}$, define $f \simeq f'$ iff `up?` $f =$ `up?` f' (and $d \simeq d'$ iff $d = d'$ for data values d, d'). Then we can use the equations of `FLAG` to show that $f \simeq f'$ implies `up` $f \simeq$ `up` f' and `dn` $f \simeq$ `dn` f' and `rev` $f \simeq$ `rev` f', and of course `up?` $f \simeq$ `up?` f'. Hence $\simeq$ is a hidden congruence on A.

Therefore we can show $A \models\!\!\!\equiv (\forall \texttt{F} : \texttt{Flag})$ `rev rev F` $=$ `F` just by showing $A \models (\forall \texttt{F} : \texttt{Flag})$ `up? rev rev F` $=$ `up? F`. This follows by ordinary equational reasoning, since `up? rev rev F` $=$ `not(not(up? F))`. Therefore the equation is behaviorally satisfied by any `FLAG`-algebra A.

It is easy to do this proof mechanically using OBJ3, since all the computations are just ordinary equational reasoning. We set up the proof by opening `FLAG` and adding the necessary assumptions; here `R` represents the candidate relation $\simeq$:

```
openr FLAG .
op _R_ : Flag Flag -> Bool .
var F1 F2 : Flag .
eq F1 R F2 = (up? F1 == up? F2) .
ops f1 f2 : -> Flag .
close
```

The new constants `f1`, `f2` are introduced to stand for universally quantified variables, following the lemma of constants, and `==` is OBJ3's builtin equality test[12]. We now show that `R` is a hidden congruence:

```
open .  eq up? f1 = up? f2 .  red (up f1) R (up f2) .  ***> should be: true
red (dn f1) R (dn f2) .  ***> should be: true red (rev f1) R (rev f2) .
***> should be: true close
```

where `red` is a command that tells OBJ to "reduce" the subsequent term, i.e., to apply equations as left-to-right rewrite rules, until a term is obtained where no rule applies.

Finally, we show that all `FLAG`-algebras behaviorally satisfy the equation with:

```
red (rev rev f1) R f1 .
```

[12]This operation reduces its two arguments, and then checks whether the results are identical. It is known that this gives equality under certain general conditions, which do hold here [37].

All the above code runs in OBJ3, and gives `true` for each reduction, provided the following lemma about the Booleans is added somewhere,

```
eq not not B = B .
```

where `B` is a Boolean variable. I think this proof is about as simple as could be hoped for. □

Just as there is a rich lore about doing inductive proofs, so more and more lore is accumulating about doing coinductive proofs. For example, the third reduction in the example above is unnecessary; however, it is more trouble to justify its elimination than it is to ask OBJ to do it; see [38].

3.2 Nondeterminism

Because nondeterminism is very problematic for ordinary algebraic specification, it is perhaps surprising that it is already an inherent facet of hidden algebra. We first illustrate this with the following very simple example:

```
th C is pr DATA .
  op c : -> Nat .
endth
```

Here `c` has some natural number value in every model, and every number can occur; each model chooses exactly one. However, there can also be arbitrary junk in models, so it makes sense to restrict to reachable models; then the choice of a value for `c` completely characterizes a model. I like to describe this by saying that each model is a "possible world" in which some fixed choice has been made for each nondeterministic possibility.

It is also easy to restrict the choice of a value for `c`, by adding an equation like one of the following:

```
eq c => 1 = true .
eq 2 => c = true .
eq odd(c) = true .
eq prime(c) = true .
eq c == 1 or c == 2 = true .
```

The last equation suggests a rather cute way to specify nondeterministic choice in hidden algebra:

```
th CH is pr DATA .
  op _|_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq N | M == N or N | M == M = true .
endth
```

Here again, models are "possible worlds," where some choice of one of `N`, `M` is made for each pair `N`, `M`. It is not hard to prove that this choice function is idempotent, i.e., satisfies the equation

```
  eq N | N = N .
```

However, the commutative and associative properties fail for some models (the reader is invited to find the appropriate models) and hence for the theory.

Neither example of nondeterminism above involves state, which is the most characteristic feature of hidden algebra, so we really should give an example of nondeterminism with a hidden sort. For some reason, vending machines are very popular for illustrating various aspects of systems, especially nondeterminism and concurrency. The spec below describes perhaps the simplest vending machine that is not entirely trivial: when you put a coin in, it nondeterministically gives you either coffee or tea, represented say by true and false, respectively; and then it goes into a new state where it is prepared to do the same again. In this spec, `init` is the initial state, `in(init)` is the state after one coin, `in(in(init))` is the state after two coins, etc., while `out(init)` is what you get after the first coin, `out(in(init))` after the second, etc.

```
th VCT is sort St .
  pr DATA .
  op in : St -> St .
  op out : St -> Bool .
endth
```

As before, it is easy to restrict behavior by adding equations like

```
  cq out(in(in(S))) = not out(S) if out(S) = out(in(S)).
```

which says that you cannot get the same substance three times in a row. (It is interesting to notice that this equation will guarantee fairness.)

For examples like this, it is also interesting to look at the final algebra F, for the signature without the constant `init`: according to the "magic formula," it consists (up to isomorphism) of all Boolean sequences – i.e., it is the algebra of (what are called) *traces*; in fact, contexts are the natural generalization of traces to a non-monadic world. Since there is a unique (hidden) homomorphism $M \to F$ for any model M of `VCT`, the image of `init` under this map characterizes the behavior of M. This simple and elegant situation holds for nondeterministic concurrent systems in general. (More information about nondeterminism and final models can be found in [38].)

The approach to nondeterminism in hidden algebra is quite different from that which is traditional in automaton theory: in hidden algebra, each possible behavior appears in a different possible world, whereas a nondeterministic automaton includes all choices in a single model. The possible worlds approach corresponds to real computers, which are always deterministic, and must simulate nondeterminism, e.g., using pseudo-random numbers. Chip makers don't make nondeterministic Turing machines or automata; if they could then $P = NP$ wouldn't be a problem!

3.3 Proving Behavioral Refinement

The simplest view of behavioral refinement assumes a specification (Σ, E) and an implementation A, and asks if $A \models\!\!\equiv_\Sigma E$; the use of behavioral satisfaction is significant here, because it allows us to treat many subtle implementation tricks that only 'act as if' correct, e.g., data structure overwriting, abstract machine interpretation, and much more.

Unfortunately, trying to prove $A \models\!\!\equiv_\Sigma E$ directly dumps us into the semantic swamp mentioned in the introduction. To rise above this, we work with a specification E' for A, rather than an actual model[13]. This not only makes the proof far easier, but it also has the advantage that the proof will apply to any other model A' that (behaviorally) satisfies E'. Hence, what we prove is $E' \models\!\!\equiv E$; in

[13] Some may object that this maneuver isolates us from the actual code used to define operations in A, preventing us from verifying that code. However, we contend that this isolation is actually an *advantage*, since only about 5% of the difficulty of software development lies in the code itself [7], with much more of the difficulty in specification and design; our approach addresses these directly, without assuming the heavy burden of a messy programming language semantics. But of course we can use algebraic semantics to verify code if we wish, as extensively illustrated in [37]. Thus we have achieved a significant separation of concerns.

semantic terms, this means that any A (behaviorally) satisfying E' also (behaviorally) satisfies E; and very significantly, it also means that we can use hidden coinduction to do the proof. The method is just to prove that each equation in E' is a behavioral consequence of E, i.e., a behavioral property of every model (implementation) of E. More details and some examples are given in [38], including the proof that a pointer with an array gives a behavioral refinement of the stack spec.

3.4 The Object Paradigm

Objects have local states with visible local "attributes" and "methods" to change state. Objects also come in "classes," which can "inherit" from other classes, and objects can communicate concurrently and nondeterministically with other objects in the same system. This paradigm has become dominant in many important application areas. We have already seen how to handle most of these features with hidden algebra. Aspects of concurrency and inheritance are treated in [33, 38]. A full treatment of inheritance requires the use of order sorted algebra for subclasses [41].

4 Summary and Related Work

This paper has presented hidden algebra as a natural next step in the evolution of algebraic specification, that can handle the main features of the object paradigm. Of course, no one would invent a method like coinduction for examples as simple as our flag example; this was chosen just to bring out the basic ideas clearly. Much more complex examples have been done, including correctness proofs for an optimizing compiler and for a novel communication protocol [50, 34]. Hidden algebra first appeared in [31], and was subsequently elaborated in papers including [33, 10]; an important precursor was work by Goguen and Meseguer on what they called "abstract machines" [39]. The rapidly growing literature on hidden algebra includes [17, 38, 59, 12, 52, 18, 60]. Coinduction seems to give proofs that are about as simple as possible, but more experience is needed before this can be said with complete certainty. The closely related area of coalgebra also uses coinduction, and also has a rapidly growing literature, including [74, 53, 54, 55]. However, it seems that coalgebra has difficulty with nondeterminism, concurrency, and operators with multiple state arguments.

It can be argued that algebraic specification is now entering a golden age, in which new techniques are bringing old goals to fruition in unexpected ways, and are also opening new horizons from which exciting new goals seem reachable. We have discussed hidden algebra and its cousin coalgebra. Another important new development is rewriting logic [61, 62], a weakening of equational logic that provides an ideal operational semantics for rapidly implementing many term rewriting algorithms [14], as well as for describing and comparing various kinds of concurrency [62]; rewriting logic has been efficiently implemented in Maude [63, 13]. The CafeOBJ system [20, 23] provides industrial strength implementations of rewriting logic, as well as of ordinary order sorted equational logic, hidden sorted equational logic, *and* all their combinations [19, 20]! The designs for both Maude and CafeOBJ are heavily indebted to that of OBJ, and indeed can be considered versions of OBJ. There is also exciting new work in term rewriting [16] (which is the basis for implementing systems like OBJ3, Maude and CafeOBJ), for example in France around Jean-Pierre Jouannoud and Adel Bouhoula, on induction and termination proofs [9], including the SPIKE and CiME systems [8], and some new work on proving behavioral properties with a similar technology [3]. The issues discussed in this paper seem to be of increasing importance for computer science, and I think we can look forward to continuing progress.

References

[1] Jean Benabou. Structures algébriques dans les catégories. *Cahiers de Topologie et Géometrie Différentiel*, 10:1–126, 1968.

[2] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer, 1980. Lecture Notes in Computer Science, Volume 81.

[3] Narjes Berregeb, Adel Bouhoula, and Michaël Rusinowitch. Observational proofs with critical contexts. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 1998.

[4] Michael Bidoit and Rolf Hennicker. Behavioral theories and the proof of behavioral properties. *Theoretical Computer Science*, 165:3–55, 1996.

[5] Michel Bidoit, Rolf Hennicker, and Martin Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming*, 25(2–3), 1995.

[6] Garrett Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

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

[8] Adel Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, 1997.

[9] Adel Bouhoula and Jean-Pierre Jouannaud. Automata-driven automated induction. In *Proceedings, 12th Symposium on Logic in Computer Science*, pages 14–25. IEEE, 1997.

[10] Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: an institutional approach. In Andrew William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 75–92. Prentice-Hall, 1994. Also Technical Report ECS-LFCS-8892-253, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[11] Graham Button and Wes Sharrock. Occasioned practises in the work of implementing development methodologies. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 217–240. Academic, 1994.

[12] Corina Cîrstea. A semantical study of the object paradigm. Transfer thesis, Oxford University Computing Laboratory, 1996.

[13] 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.*

[14] 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.

[15] CoFI. CASL summary, 1998. `http://www.brics.dk/Projects/CoFi/`.

[16] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewriting systems. In *Handbook of Theoretical Computer Science, Volume B*, pages 243–309. North-Holland, 1990.

[17] 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*.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] Marie-Claude Gaudel and Igor Privara. Context induction: an exercise. Technical Report 687, LRI, Université de Paris-Sud, 1991.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] Joseph Goguen. Proving and rewriting. In Hélène Kirchner and Wolfgang Wechler, editors, *Proceedings, Second International Conference on Algebraic and Logic Programming*, pages 1–24. Springer, 1990. Lecture Notes in Computer Science, Volume 463.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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. Nomuzu, Japan, April 1998.

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

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

[38] 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.

[39] 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.

[40] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985. Preliminary versions have appeared in: *SIGPLAN Notices*, July 1981, Volume 16, Number 7, pages 24–37; SRI Computer Science Lab, Report CSL-135, May 1982; and Report CSLI-84-15, Center for the Study of Language and Information, Stanford University, September 1984.

[41] 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.

[42] 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.

[43] 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).

[44] 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.

[45] 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.

[46] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Abstract data types as initial algebras and the correctness of data representations. In Alan Klinger, editor, *Computer Graphics, Pattern Recognition and Data Structure*, pages 89–93. IEEE, 1975.

[47] 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. An early version is "Initial Algebra Semantics", by Joseph Goguen and James Thatcher, IBM T.J. Watson Research Center, Report RC 4865, May 1974.

[48] 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.* World Scientific, to appear. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.

[49] John Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, 1975. Computer Science Department, Report CSRG–59.

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

[51] Rolf Hennicker. Context induction: a proof principle for behavioural abstractions. *Formal Aspects of Computing*, 3(4):326–345, 1991.

[52] Shusaku Iida, Michihiro Matsumoto, Răzvan Diaconescu, Kokichi Futatsugi, and Dorel Lucanu. Concurrent object composition in CafeOBJ. Technical Report IS–RR–96–0024S, Japan Advanced Institute for Science and Technology, 1997.

[53] Bart Jacobs. Objects and classes, coalgebraically. In B. Freitag, Cliff Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer, 1996.

[54] Bart Jacobs. Invariants, bisimulations and the correctness of coalgebraic refinements. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, pages 276–291. Springer, 1997. Lecture Notes in Computer Science, Volume 1349.

[55] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, June 1997.

[56] 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.

[57] Saunders Mac Lane and Garrett Birkhoff. *Algebra.* Macmillan, 1967.

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

[59] Grant Malcolm. Behavioural equivalence, bisimilarity, and minimal realisation. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specifications: 11th Workshop on Specification of Abstract Data Types*, pages 359–378. Springer Lecture Notes in Computer Science, Volume 1130, 1996. (Oslo Norway, September 1995).

[60] Michihiro Matsumoto and Kokichi Futatsugi. Test set coinduction: Toward automated verification of behavioural properties. In *Proceedings of the Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier Science, to appear 1998.

[61] 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.

[62] José Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[63] 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.

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

[65] 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.

[66] 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.

[67] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[68] Peter Padawitz. Towards the one-tiered design of data types and transition systems. In *Proceedings, WADT'97*, pages 365–380. Springer, 1998. Lecture Notes in Computer Science, Volume 1376.

[69] Peter Padawitz. Swinging types = functions + relations + transition systems, 1999. Submitted to *Theoretical Computer Science.*

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

[71] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association for Computing Machinery*, 15:1053–1058, 1972.

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

[73] Horst Reichel. Behavioural validity of conditional equations in abstract data types. In *Contributions to General Algebra 3*. Teubner, 1985. Proceedings of the Vienna Conference, June 21-24, 1984.

[74] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

[75] Grigore Roşu and Joseph Goguen. Hidden congruent deduction. In Gernot Salzer, editor, *Proceedings, 1998 Workshop on First Order Theorem Proving*. Johannes Kepler Univ. Linz, 1998.

[76] Bartel van der Waerden. *A History of Algebra*. Springer, 1985.

[77] Steven Zilles. Abstract specification of data types. Technical Report 119, Computation Structures Group, Massachusetts Institute of Technology, 1974.

A OBJ3 Output

Below is the output that OBJ3 produces when it executes this paper (there is a little program that extracts the executable code from the paper, and passes it to OBJ3 for execution; the source file for this paper has two "invisible" OBJ3 modules, `EVENAT` and `DATA`, which are needed to make others work):

```
             \||||||||||||||||||/
           --- Welcome to OBJ3 ---
             /||||||||||||||||||\
 OBJ3 version 2.04oxford built: 1994 Feb 28 Mon 15:07:40
    Copyright 1988,1989,1991 SRI International
           1998 Nov 8 Sun 10:21:48

==========================================
th AUTOM
==========================================
obj NATP
==========================================
***> This is the invisible EVENAT module:
==========================================
obj EVENAT
==========================================
obj NFLAG
==========================================
***> This is the invisible DATA module:
==========================================
obj DATA
```

```
==========================================
th FLAG
==========================================
***> prove rev rev F = F :
==========================================
openr FLAG
==========================================
op _ R _ : Flag Flag -> Bool .
==========================================
var F1 F2 : Flag .
==========================================
eq F1 R F2 = ( up? F1 == up? F2 ) .
==========================================
ops f1 f2 : -> Flag .
==========================================
close
==========================================
open
==========================================
eq up? f1 = up? f2 .
==========================================
reduce in FLAG : up f1 R up f2
rewrites: 4
result Bool: true
==========================================
***> should be: true
==========================================
reduce in FLAG : dn f1 R dn f2
rewrites: 4
result Bool: true
==========================================
***> should be: true red (rev f1) R (rev f2) .
==========================================
***> should be: true close
==========================================
reduce in FLAG : rev (rev f1) R f1
rewrites: 7
result Bool: true
==========================================
th C
==========================================
th CH
==========================================
th VCT
OBJ> Bye.
```

The `true` results above indicate that OBJ3 has in fact done the computations that constitute the proof.