Math. Struct. in Comp. Science (1999), *vol.* 9, *pp.* 287–319. Printed in the United Kingdom

Hidden coinduction: behavioural correctness proofs for objects

JOSEPH A. GOGUEN[†] and GRANT MALCOLM[‡]

[†] *Department of Computer Science and Engineering, University of California at San Diego, USA.*

[‡] *Connect, Department of Computer Science, University of Liverpool, UK.*

Received 27 November 1997; revised 10 December 1998

This paper unveils and motivates an ambitious programme of hidden algebraic research in software engineering. We begin with an outline of our general goals, continue with an overview of results, and conclude with a discussion of some future plans. The main contribution is powerful *hidden coinduction* techniques for proving behavioural correctness of concurrent systems, and several mechanical proofs are given using OBJ3. We also show how modularization, bisimulation, transition systems, concurrency and combinations of the functional, constraint, logic and object paradigms fit into hidden algebra.

1. Introduction

The initial goal of our hidden research was both straightforward and ambitious:

(A1) To give a semantics for software engineering, and for the object paradigm in particular, supporting correctness proofs that are as simple and mechanical as possible.

This emphasis on the *effectiveness of proofs* seemed to rule out approaches based on models expressed in set theory, denotational semantics, higher order logic, type theory, *etc.*, because of the difficulties of proving things in these approaches. An *equational* approach seemed worth exploring, because equational logic achieves maximal simplicity and mechanization, while still allowing full expressiveness. So building on a long tradition in computing science (often called *algebraic specification*), we use algebra for our semantics. However, the hidden algebra approach is distinct in that it uses sorts in two different ways:

1 some sorts are used for data values (for example, of attributes), as in the algebraic approach to data types; and
2 some sorts are used for states, as in the algebraic approach to abstract machines.

The latter gives us objects, classes, *etc.*.

These two uses of sorts are *dual*. Induction is used to establish properties of data types, whereas coinduction is used to establish properties of objects. Similarly, initiality is important for data types, whereas finality is important for states. However, we do not insist that implementations of data types must be initial, or that implementations of abstract machines (*i.e.*, objects) must be final; on the contrary, we accept any implementation that satisfies the given axioms. This is important because, in general, the best implementations are neither initial nor final, but somewhere between.

The hidden paradigm takes as basic the notion of *behavioural abstraction,* or more precisely, *behavioural satisfaction*: our specifications characterize how objects (and systems) behave, not how they are implemented; they provide a notion of *behavioural type*, which we prefer to call a *hidden theory* (or *behavioural theory* or *hidden specification*). Our correctness proofs show that one hidden theory behaviourally satisfies another, in the sense that any model of the second theory yields a model of the first. This principle allows us to justify our proof techniques model theoretically, reflecting our view that

(S) semantics is fundamental at the meta level (for showing correctness for proof rules), while syntax is fundamental at the object level (for building, describing and verifying systems).

(See Goguen (1998) for further discussion and examples of this viewpoint.)

Since we use hidden sorts to specify classes of objects, order sorted algebra provides a very natural way to handle multiple inheritance; it also allows us to specify partial functions, non-terminating systems, subtypes of various kinds, error definition and recovery, coercions, overwriting, multiple representations, and more (see Goguen and Diaconescu (1994a) for a discussion of polymorphism, dynamic binding and overwriting, and Goguen and Meseguer (1987a) for a discussion of errors, coercions, *etc.*). However, for expository simplicity, this paper only treats hidden many sorted algebra. The module system of parameterized programming gives us other forms of inheritance, plus all the power of higher order functional programming in a first order setting, which facilitates both proving and programming (Goguen 1990a).

Modularization is weak in many contemporary languages, especially in object and logic languages; moreover, it can be surprisingly hard to reuse code in practice, and development methods to support reuse remain vague. Therefore, we broadened our research goals to

(A2) develop powerful modularization techniques and semantically sound methods for refinement (dealing with hierarchies of abstract machine implementations).

We later realized that constraints were already inherent in our approach and that an entirely new style of logic programming could develop out of hidden algebra by adding existential quantifiers. This put a new item on our hidden agenda:

(A3) to combine the functional, object, logic, concurrent and constraint paradigms.

What is perhaps amazing is that it is actually *artificial* to exclude any of these paradigms: concurrency, nondeterminism, local states, classes, inheritance, constraints, streams, existential queries, *etc.* are all very natural parts of the hidden world.

In summary, we are concerned with semi-mechanical proof techniques for hidden (order sorted) algebra, as a way of proving behavioural properties of systems, including refinement. We make the no doubt outrageous claim that our hidden approach leads to simpler proofs than other formalisms; this is because we exploit algebraic structure that most other approaches discard. The following sections give an overview of what support we now have for this claim.

The next two subsections provide further motivation for hidden algebra and compare our research programme to some others. Section 2 gives an introduction to hidden algebra, and Section 3 discusses hidden coinduction for proving behavioural properties. Section 4 shows how our techniques can be used to prove correctness of refinements (*i.e.*, implementations), and Section 5 discusses composite systems of concurrent interacting objects. Many of the results we present are new. Section 6 gives some conclusions and directions for future research, and Appendix A gives two further small mechanical proofs.

The examples in this paper use OBJ3 to express theories and do proofs. We give 'proof scores', in which humans have done the interesting structuring work, while OBJ does the boring computations. Of course, much of the 'interesting' work could also be automated, including the examples in this paper, but this is certainly impossible in general. We have chosen the balance between human and machine efforts to enhance the readablity of the proof scores. We assume some familiarity with basic many sorted algebra and with OBJ. The relevant background (and much more!) can be found in Goguen and Malcolm (1996), Goguen (1998), Goguen *et al.* (1978), Meseguer and Goguen (1985) and Goguen *et al.* (1998), among many other places. For ease of reference, formal textual items (theorems, definitions, *etc.*) are numbered sequentially on the same counter.

2. Hidden algebra

Hidden algebra captures the fundamental distinction between data values and internal states by modelling the former with 'visible' sorts and the latter with 'hidden' sorts. These are treated in the next two subsections, respectively.

2.1. *Visible data values*

The components of a system must use the same representations for the data that they share, otherwise they cannot communicate; thus it makes sense to declare a fixed collection of shared data values, bundled together in a single algebra.

Definition 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. For convenience, we assume that $D_v \subseteq \Psi_{[],v}$ for each $v \in V$. We may also call (V, Ψ, D) the **visible data universe**, and we call V the **visible sorts**.

In practice, there may be multiple representations for data with translations among them, and representations may change during development; but our simplifying assumption can easily be relaxed.

The above definition deals with 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*, which also supports proofs by induction. More precisely, the 'no junk' half of initiality validates induction proofs over any reachable algebra, while the 'no confusion' half supports disequality proofs. For example, the following OBJ3 specification for the natural numbers is used in later examples:

```
obj NAT is sort Nat .
  op  0 : -> Nat .
  op s_ : Nat -> Nat [prec 1].
  op _<_ : Nat Nat -> Bool .
  var N M : Nat .
  eq   0 < s N = true .
  eq   N <   N = false .
  eq s N <   0 = false .
  eq s N < s M = N < M .
endo
```

Here `NAT` is the name of the module and `Nat` is the name of the sort for natural numbers. The keyword pair `obj...endo` indicates initial algebra semantics. The underbar characters indicate where an argument goes, so that the successor operator `s_` has prefix syntax, and the inequality operator `_<_` has infix syntax. The rest of the OBJ3 syntax used here should be fairly self-evident; for more on OBJ3, see Goguen and Malcolm (1996) and Goguen *et al.* (1998).

Our examples assume a fixed OBJ3 module `DATA` giving a signature and axioms for D (D need not be a term model for `DATA`, or even an initial model). The following says that `DATA` is just the natural numbers; this is adequate for (most of) this paper, noting that `NAT` imports the Booleans.

```
obj DATA is
  pr NAT .
endo
```

Of course, the (cumulative) signature of `DATA` must be Ψ, and we let F denote its set of equations.

In writing a specification for the data involved in a problem, it is encouraging to recall that any computable algebra can be finitely specified with initial algebra semantics, and that data types used as values really should be computable. However, one could just as well use a loose semantics, by explicitly giving any properties that are needed, and then noting that any Ψ-algebra D satisfying these properties could be the data universe. Indeed, some data universes are not computable, for example, the real and complex numbers, which are important for constraint logic programming. However, even these data types can be captured with initial algebra semantics by using an uncountable number of constants and equations (though order sorted algebra should be used to prohibit division by zero).

2.2. *Hidden signatures and hidden algebras*

This subsection gives some basic definitions for hidden algebra. Hidden algebra uses a loose behavioural semantics over a fixed data algebra. Unlike the case in Orejas *et al.* (1996), there is no competition between this and initial algebra semantics, because they are used for different purposes.

Definition 2. A **hidden signature (over** (V, Ψ, D)**)** is a pair (H, Σ), where H is a set of **hidden sorts** disjoint from V and 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 **behavioural**) **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 even E for short.

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. Note that every operation in a hidden signature is either a method, an attribute, or a constant. 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. Note that this definition allows multiple hidden sorts: these are useful for (in the jargon of the object paradigm) *complex attributes*, which are class valued attributes; however, it is always possible to reduce to a single hidden sort without loss of expressiveness, at some cost in complexity.

The following example may help clarify this definition. This code uses OBJ3 syntax for theories, where the keyword pair `th...endth` with ‘`pr DATA`’ indicates a loose semantics ‘protecting’ `DATA` – we will explain this soon.

Example 3. We specify 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 .
   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 operations `up`, `dn` and `rev` are methods to change the state of flag objects, and `up?` is an attribute that tells whether or not the flag is up. All these operations have prefix syntax.

We could add a hidden constant `newf` as an initial state for `FLAG` objects, with an equation

```
eq up? newf = false .
```

to set the initial value of the attribute. Of course, the initial value could equally well be `true`, or it could be left undefined by simply not giving such an equation.

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.

Definition 4. Given a hidden signature (H, Σ), a **hidden Σ-algebra** A is a (many sorted) Σ-algebra A such that $A\lceil_\Psi = D$. A **hidden Σ-homomorphism** is a (many sorted) Σ-homomorphism that is the identity on visible sorts.

We next define behavioural satisfaction of an equation; intuitively, its two terms should '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 5. Given a hidden signature (H, Σ) and a sort s, a **Σ-context** of sort s is a visible sorted Σ-term having a single occurrence of a new variable symbol z of sort s. 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, and let $C_\Sigma[z]$ denote the V-indexed set of contexts with hidden variable z.

A hidden Σ-algebra A **behaviourally 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 $A \models_\Sigma (\forall X)\ t = t'$, and we may drop the subscript Σ.

Similarly, A **behaviourally satisfies** a conditional equation e of the form

$$(\forall X)\, t = t' \textbf{ if } t_1 = t'_1, ..., t_m = t'_m$$

iff for every assignment $\theta : X \to A$, we have

$$\theta^*(c[t]) = \theta^*(c[t'])$$

for all appropriate contexts c whenever

$$\theta^*(c_j[t_j]) = \theta^*(c_j[t'_j])$$

for $j = 1, ..., m$ and all appropriate contexts c_j (here θ^* denotes the unique Σ-homomorphic extension of θ). As with unconditional equations, we write $A \models_\Sigma e$.

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

Finally, a hidden Σ-algebra A is **reachable** iff the unique Σ-homomorphism from the initial (term) Σ-algebra T_Σ is surjective.

Example 6. The following are some contexts for `FLAG`:

$$\begin{aligned} c_1[z] &= \texttt{up?}\ z \\ c_2[z] &= \texttt{up? up rev}\ z \\ c_3[z] &= \texttt{up? dn dn}\ z \end{aligned}$$

There are an infinite number of contexts; all begin with `up?` since that is the only attribute.

We next give some models for the `FLAG` theory, that is, some objects of class `Flag`.

Example 7. Let us first 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 from $\{up, dn, rev\}$, 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 behavioural 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. However, it is *behaviourally* satisfied by both models. This shows that behavioural satisfaction is often more appropriate for computing science applications. (We will later use coinduction to prove that every `FLAG`-model behaviourally satisfies this equation.)

Example 8. The following OBJ3 code specifies a cell that holds a single natural number:

```
th X is sort State .
  pr DATA .
  op putx : Nat State -> State .
  op getx_ : State -> Nat .
  var S : State .  var N M : Nat .
  eq getx putx(N,S) = N .
endth
```

Objects of class `X` are really just 'program variables' of integer type, that is, cells that hold an integer, and the models are ways of implementing such cells.

The following proposition justifies the simplest ways to reason about behavioural satisfaction.

Proposition 9. In proving $E \models e$, the ordinary rules of equational deduction are valid, including substituting one behavioural equation into another, and, of course, symmetry and transitivity; visible equations can also be used in such proofs.

This result is easy to prove, and can be very useful. For example, if we want to prove

```
getx putx(N, putx(M, S)) = N ,
```

for the above theory `X`, we can just do the following:

```
red getx putx(N, putx(M, S)) == N .
```

(OBJ3 complains about the variables, but does the reduction anyway, treating them as constants, and giving the correct result, `true`.) However, something more powerful than reduction is needed to prove the equation about double reverse given above, or to prove

```
putx(M, putx(N, S)) = putx(M, S) .
```

Unfortunately, it is easy to write theories that have no models. For example, if we add a constant `newf` of sort `Flag` and the equation

```
    eq not up? F = up? F .
```

to `FLAG`, we can prove that `true` = `false`, which contradicts Definition 4. This motivates the following definition.

Definition 10. A hidden theory is **consistent** iff it has at least one model.

Some techniques for guaranteeing consistent specifications from Goguen *et al.* (1998) are summarized in Theorem 13 below, which uses the following concepts.

Definition 11. A $\Psi(X)$-term is **local** iff it is a constant or a variable (that is, is in D or in X); a $\Sigma(X)$-term that is not a $\Psi(X)$-term is **local** iff all visible proper subterms are constants or variables. Let $L_{\Sigma,s}$ denote the set of local ground Σ-terms of sort s. An equation is **local** iff its left-hand and right-hand sides are local and its conditions (if any) are $\Psi(X)$-terms; a set of equations is local iff each one is. A **constraint** is an equation such that both its terms have their top operations in Ψ.

A local equation cannot be a constraint. Constraints constrain the values of undefined terms over a theory, as discussed in some detail in Section 2.3, which shows how this relates to nondeterminism.

Definition 12. A set E of Σ-equations is D-**complete** iff $D \models_\Psi (\forall\emptyset)\, t = t'$ implies $E \models_\Sigma (\forall\emptyset)\, t = t'$ for all Ψ-terms t and t'.

Theorem 13. If the equations E in a hidden theory are D-complete and Church–Rosser local rewrite rules, the theory is consistent.

Proof. We show consistency by constructing a model M with carriers $M_h = L_{\Sigma,h}$ for $h \in H$ and $M_v = D_v$ for $v \in V$. Its methods are interpreted as term building operations, noting that this preserves locality. For an attribute $\sigma \in \Sigma_{hw,v}$, given $t \in M_h$ and $d \in D_w$, let $M_\sigma(t,d) = d'$ if $E \models \sigma(t,d) = d'$, and otherwise let $M_\sigma(t,d)$ be some element d_v of D_v, fixed for each visible sort v. Because E is Church–Rosser, there is at most one d' with $E \models \sigma(t,d) = d'$, so M_σ is well-defined.

To show $M \models\!\!\!\models E$, let $(\forall X)\, t = t'$ **if** $t_1 = t'_1, \ldots, t_m = t'_m$ be in E, and let $\theta : X \to M$. Because the unique Σ-homomorphism $g : T_\Sigma \to M$ is surjective, there is $\phi : X \to T_\Sigma$ with $\theta^* = \phi^* ; g$. Suppose $\theta^* t_i = \theta^* t'_i$ for $i = 1, \ldots, m$, then because t_i, t'_i are $\Psi(X)$-terms, we have $D \models_\Psi (\forall\emptyset)\, \phi^* t_i = \phi^* t'_i$, and by D-completeness, this implies $E \models_\Sigma (\forall\emptyset)\, \phi^* t_i = \phi^* t'_i$, so

$$E \models_\Sigma (\forall\emptyset)\, \phi^* t = \phi^* t' \ . \qquad (1)$$

By Proposition 20 below, we can show behavioural satisfaction by considering only contexts that are local terms. Since t is a local term, so too is $c[t]$ for any local context c, and by the definition of M it follows that

$$\theta^* c[t] = g(\phi^* c[t]) = \begin{cases} d' & \text{if } E \models_\Sigma (\forall\emptyset)\, \phi^* c[t] = d' \\ d_v & \text{otherwise.} \end{cases}$$

If there is some d' with $E \models_\Sigma (\forall\emptyset)\, \phi^* c[t] = d'$, then by (1) it follows that $\theta^* c[t] = d' = \theta^* c[t']$. Similarly, if there is no such d', then $\theta^* c[t] = d_v = \theta^* c[t']$. In both cases $\theta^* c[t] = \theta^* c[t']$, so $M \models\!\!\!\models E$ as desired. □

Many examples in this paper can be shown consistent using this result. A sufficient condition for the Church–Rosser property is that the equations are nonoverlapping; and for conditional equations, the left-hand sides may overlap provided the conditions are disjoint. Once a specification has been shown consistent ignoring its nonlocal equations, the consistency of constraints can be considered separately, though determining whether a set of constraints has a solution can be arbitrarily difficult, even unsolvable.

Example 14. This hidden theory for arrays is used in a later example, and can be seen to be consistent by using Theorem 13:

```
th ARR is sort Arr .
  pr  DATA .
  op  nil : -> Arr .
  op  put : Nat Nat Arr -> Arr .
  op  _[_] : Arr Nat -> Nat .
  var I J N : Nat .  var A : Arr .
  eq  nil[I]  =  0 .
  cq  put(N,I,A)[J] = N if I == J .
  cq  put(N,I,A)[J] = A[J] if not I == J .
endth
```

Here `nil` is the empty array, `A[I]` is the value of `A` at index `I`, and `put(N,I,A)` puts `N` at `I` in `A`. There are no hidden equations.

Hidden algebra is related to the constraint logic as described in Diaconescu (1994) and Diaconescu (1996): (V, Ψ) is the signature of built-ins, D the model of built-ins, and Σ is an extension of the 'logical' signature. However, hidden algebras differ from constraint logic models because the built-ins are protected.

2.3. *Nondeterminism*

Modern distributed programming paradigms cannot do without nondeterminism, because the nodes of a network cannot be expected to know what the other nodes are going to be doing. Therefore it is essential that a formalism intended to be useful for modern software engineering should treat nondeterminism in a simple and natural way. However, most concurrency calculi treat nondeterminism in complex and unnatural ways, and, moreover, there are sharp ongoing debates among the advocates of the various approaches, with no obvious resolution in sight.

Nondeterminism is inherent to the hidden paradigm; it arises whenever some attribute values are not determined by a specification. To understand this, it may help to view models as 'possible worlds,' where each possible combination of nondeterministic choices appears in a different world. However, this does not mean more than one value can occur in a given world; on the contrary, each model is deterministic, in that attributes only take one value at a time. However, a given hidden specification may have multiple models, in which the attributes have completely different values.

Definition 15. Given a hidden theory $P = (H, \Sigma, E)$, a ground Σ-term t is **defined** iff for every context c (of appropriate sort), there is some $d \in D$ such that $E \models c[t] = d$, otherwise, t is **undefined**. P is **lexic** iff all ground terms are defined.

Fact 16. Given a hidden theory $P = (H, \Sigma, E)$,

1 A visible term t is defined iff $E \models t = d$ for some $d \in D$.
2 P is lexic iff all visible ground terms are defined.
3 If we call a ground term **invisible** iff it is of hidden sort and has no contexts; then all invisible terms are defined.
4 P is lexic if it has no hidden (generalized) constants.

The property of having no undefined ground terms corresponds to Guttag's notion of sufficient completeness (Guttag 1977). However, not only do we not require this condition, but we claim that undefined terms are very useful in system development, and even at run time. Instead of having explicitly to say something is 'undefined', one simply does not define it; then it can have any value consistent with the given theory, and indeed, all possible combinations of values occur among the models of the theory. Hidden nondeterminism avoids theological disputes, for example, between angelic and demonic nondeterminism; we simply get a certain range of implementation freedom, *i.e.*, of possible worlds.

Example 17. Consider the following simple theory with one hidden sort, one natural number valued attribute, one hidden constant, no equations, and the usual data (naturals and Booleans):

```
th EX1 is sort H .
  pr DATA .
  op c :   -> H .
  op a : H -> Nat .
endth
```

There is exactly one undefined visible ground term here, namely `a(c)`. Hence this theory calls for a nondeterministic choice of a natural number, and indeed (up to behavioural equivalence, as defined in Section 3) there is exactly one reachable Σ-algebra for each choice of a natural number for the attribute. There are also infinitely many non-reachable models; these worlds may have arbitrarily many other 'unnamed' (*i.e.*, unreachable or 'junk') objects, each with a natural number attribute. If we add the constraint

```
eq a(H) < s s s s 0 = true .
```

the nondeterminism is restricted so that (again up to behavioural equivalence) there are just four reachable models, each a world where the attribute of the object `c` has value 0, 1, 2 or 3. The unreachable models contain other objects, each of which has an attribute with value 0, 1, 2 or 3.

Things get more interesting when there are methods as well as attributes. Then the elements reachable from a given element of a hidden algebra are the *states* that can arise by applying methods to that element; a connected component of elements consists of all states for a single object. It is almost obligatory to test drive a new specification technology over a range of stacks, because most approaches have already done so; hence stacks are a convenient (but minimal) benchmark for comparing approaches. We first specify a non-deterministic stack. (Since this paper is limited to many sorted algebra, the handling of errors is weak; Goguen and Diaconescu (1994a) and Goguen and Meseguer (1992) show how to do it better with order sorted algebra.)

Example 18. Here the operation `push` nondeterministically puts a new natural number on top of a stack. This single operation thus corresponds to countably many nondeterministic transitions in a traditional state transition system.

```
th NDSTACK is sort Stack .
  pr DATA .
  op  empty : -> Stack .
  op  push_ : Stack -> Stack .
  op  top_  : Stack -> Nat .
  op  pop_  : Stack -> Stack .
  var S : Stack .
  eq  pop push S =  S .
  eq  pop empty  =  empty .
endth
```

Terms like `top push empty` are undefined, *i.e.*, nondeterministic, and can take any value. Each model of this specification is deterministic, and represents one possible way of resolving the nondeterminism.

Behavioural satisfaction of the first equation implies that whatever number is pushed on a stack stays there until it is popped. For example, it follows that

```
top pop push S  =  top S
```

and that

```
top pop pop push push S  =  top S .
```

However, it is *not* true that

```
top push pop S  =  top S ,
```

because the new number pushed on `S` may be different from the old one.

The term `top empty` is also undefined, and hence can take any value. Of course, we could fix its value with an equation like

```
top empty  =  0 .
```

Moreover, we could constrain `push` to just one of the four values 0, 1, 2, 3 by adding an equation like that in Example 17:

```
top push S < s s s s 0  =  true .
```

It is also possible to have several different nondeterministic `push` methods, each subject to different constraints.

Models are deterministic in that operations are interpreted as functions on the carrier sets, but also in this example in that the equation

```
top push pop push empty = top push empty
```

is satisfied by all models. This says that if a value is nondeterministically pushed on to the empty stack, the stack is then popped and a value is pushed on again, then the second value is always the same as the first value pushed on to the stack. A less restrictive form of nondeterminism, in which the above equation need not be satisfied by all models, can be admitted by limiting the number of contexts that determine behavioural equivalence, as in Roşu and Goguen (1998).

This example shows that hidden semantics differs sharply from initial semantics, where terms like `top empty` would appear as new elements of sort `Nat`; it also differs from pure loose semantics, where such terms could be either new elements or old data values.

Hidden algebraic nondeterminism can be used much as in the concurrent constraint paradigm: a specification describes the possible states of an object in isolation, but what states actually occur is co-determined with other objects through their interactions, expressed as constraints. For example, the specifications for an array and a pointer into it describe all their possible states separately, but when they are put together to implement a stack, many states are no longer reachable: details of this implementation are given in Example 35 – in that context, it is impossible for the array containing all 1's to occur. Thus, hidden algebra is naturally nondeterministic; we will see that it is also well suited to concurrent, reactive systems.

Example 19. Here is a hidden version of the traditional stack theory with a non-unary deterministic `push`:

```
th STACK is sort Stack .
  pr  DATA .
  op  empty : -> Stack .
  op  push : Nat Stack -> Stack .
  op  top_ : Stack -> Nat .
  op  pop_ : Stack -> Stack .
  var S : Stack .  var I : Nat .
  eq  top push(I,S)   =  I .
  eq  pop empty   =  empty .
  eq  pop push(I,S)   =  S .
endth
```

Here `top empty` is the only undefined ground term (up to equality).

A similar situation arises if we delete the equation `nil[I] = 0` from the `ARR` theory in Example 14; then a user of an implementation of this specification may find 'garbage' in the array, and so must be careful not to rely on its having any particular initial value.

Undefined values obstruct initial hidden algebras, as shown in Theorem 22 below. Recalling that $L_{\Sigma,s}$ denotes the set of local ground Σ-terms of sort s, note that any Σ-algebra M induces a hidden Σ-algebra structure on L_Σ, which we denote L_M, by interpreting methods as term building operations, and interpreting an attribute $\sigma \in \Sigma_{w,v}$ by $(L_M)_\sigma(\ell) = M_\sigma(h_M(\ell)) = h_M(\sigma(\ell))$ for suitable $\ell \in (L_\Sigma)_w$, where h_M is the unique Σ-homomorphism $T_\Sigma \to M$. Restricting h_M to local terms gives a unique hidden Σ-homomorphism $L_M \to M$, which we denote φ_M.

Proposition 20. Let $L_\Sigma[z]_s \subseteq C_\Sigma[z]_s$ denote the set of local Σ-contexts of sort s involving the variable z of hidden sort. Then a hidden Σ-algebra M behaviourally satisfies a hidden equation $(\forall X)\ t = t'$ iff it satisfies $(\forall X)\ c[t] = c[t']$ for every visible local context $c \in L_\Sigma[z]$.

Proof. The 'only if' direction follows from the fact that every local context is a context. For the converse, note that the unique Σ-homomorphism $T_\Sigma \to L_M$ extends to a Σ-homomorphism $\widehat{\_}\colon\ T_\Sigma(z) \to L_M(z)$ that translates a context c to a local context $\widehat{c}$.

(This extension is most easily explained using a little categorical magic. To the signature inclusion $\Sigma \subseteq \Sigma(z)$ corresponds a forgetful functor from $\Sigma(z)$-algebras to Σ-algebras. This functor has a left adjoint that forms the free $\Sigma(z)$-algebra $M(z)$ over any Σ-algebra M, and that extends Σ-homomorphisms to homomorphisms of $\Sigma(z)$-algebras.) We now show that for any context c and substitution $\theta : X \to M$, we have $\theta^*(\widehat{c}[t]) = \theta^*(c[t])$. Any Σ-algebra M becomes a $\Sigma(X)$-algebra by picking values $m \in M$ of the right sort for each $x \in X$ via some assignment $\theta : X \to M$. In particular, any $t \in T_\Sigma(X)$ of the same sort as z will make $T_\Sigma(X)$ into a $\Sigma(X \cup \{z\})$-algebra, and we let $\_[t] : T_\Sigma(X \cup \{z\}) \to T_\Sigma(X)$ denote the unique Σ-homomorphism. Now $\theta : X \to M$ and t give $\theta_t : X \cup \{z\} \to M$, sending $x \in X$ to $\theta(x)$ and z to $\theta^*(t)$, which induces a $\Sigma(X \cup \{z\})$ structure on M and hence a unique Σ-homomorphism $\theta_t^* : T_{\Sigma(X\cup\{z\})} \to M$ extending θ_t. But each side of the equation is the value of c along one such homomorphism, and hence they are equal. This means M interprets a context c the same way it interprets $\widehat{c}$. Therefore if the left-hand and right-hand sides of an equation are equal in all local contexts, then for any context c, they are equal in the local context $\widehat{c}$, and therefore in c. □

Proposition 21. For a given hidden signature Σ, we have the following:

1 For any hidden Σ-homomorphism $f : M \to N$ and equation e, if $N \models e$, then $M \models e$.
2 For any Σ-algebra M and equation e, if $M \models e$, then $L_M \models e$.
3 If a hidden theory has an initial model, that initial model behaviourally satisfies any equation behaviourally satisfied by any other hidden model of the theory.
4 If either e is a ground equation or M is reachable, then $M \models e$ iff $L_M \models e$.
5 If there is a hidden Σ-homomorphism $f : M \to N$, then $L_M = L_N$.

Proof. For the first assertion, let e be of the form $(\forall X)\, t = t'$, and let $\theta : X \to L$. Then $\theta; f : X \to N$, and $N \models e$ implies $(\theta; f)^*(c[t]) = (\theta; f)^*(c[t'])$ for any context c, which implies that $f(\theta^*(c[t])) = f(\theta^*(c[t']))$, which because f is the identity on visible elements, then implies $\theta^*(c[t]) = \theta^*(c[t'])$, as desired. A similar proof can be given for conditional equations.

The proofs of the second and third assertions use the homomorphism $\varphi_M : L_M \to M$.

The fourth assertion follows by factoring assignments $X \to T_\Sigma$ through L_M.

For the fifth assertion, note first that $h_N = h_M; f$. Then $(L_M)_\sigma(\ell) = h_M(\sigma(\ell))$ and $(L_N)_\sigma(\ell) = h_N(\sigma(\ell))$. But $h_N(\sigma(\ell)) = f(h_M(\sigma(\ell)))$, and therefore $h_N(\sigma(\ell)) = h_M(\sigma(\ell))$, because f is the identity on visible sorts. □

Theorem 22. A hidden theory $P = (H, \Sigma, E)$ has an initial model, denoted L_P, iff it is consistent and lexic.

Proof. We first show that if P is consistent and lexic, then $L_M = L_N$ for any two P-models M, N. Lexicality implies that for any visible ground term t there is a unique $d_t \in D$ with $E \models (\forall \emptyset)\, t = d_t$. Then $L_M = L_N$ because $(L_M)_\sigma(\ell) = h_M(\sigma(\ell)) = d_t$, and similarly for $(L_N)_\sigma(\ell)$. Because E is consistent, there is at least one model, say M. So we have L_M and a unique homomorphism $\varphi_M : L_M \to M$. Moreover, for any model N, there is a unique homomorphism $\varphi_N : L_M = L_N \to N$. Therefore L_M is an initial model.

Conversely, if there is an initial model M, then E is consistent. For every visible term t, there is a data value $f(t)$ given by the homomorphism $f : T_\Sigma \to M$. Moreover, for any other model N with homomorphism $g : T_\Sigma \to N$, we have $g(t) = \varphi(g(t)) = f(t)$, where φ

is the unique hidden homomorphism $M \to N$. Therefore $N \models (\forall\emptyset)\ t = f(t)$. Because N is arbitrary, for every ground term t there is a data value $f(t)$ such that $E \models (\forall\emptyset)\ t = f(t)$. □

System development consists in part of progressively reducing implementation freedom[†], which may involve reducing nondeterminism, among other things. Reducing nondeterminism is consistent with software engineering practice, where all the operations in a program are deterministic, but at a given development stage many programs may still satisfy the specification. Thus, hidden nondeterminism is more appropriate for refinement (see Section 4) than the forms usually found, for example, in process algebra. Nondeterminism can also remain right down to the implementation level, where any consistent value may be returned. For example, a set of constraints may be resolved only at run time, and in different ways at different times. Thus, the same notion of nondeterminism is useful for implementation freedom and for runtime choice.

3. Behaviour and hidden coinduction

Induction is a standard technique for proving properties of initial (or more generally, reachable) algebras of a theory. Principles of induction can be justified from the fact that an initial algebra has no proper subalgebras (for example, Goguen (1998) and Meseguer and Goguen (1985)). We will see that final (terminal) algebras play an analogous role in justifying reasoning about behavioural properties with hidden coinduction. We first need the following definition.

Definition 23. Given a hidden signature Σ, a hidden subsignature $\Phi \subseteq \Sigma$, and a hidden Σ-algebra A, then **behavioural Φ-equivalence** on A, denoted $\equiv_\Phi$, is defined as follows for $a, a' \in A_s$:

$$a \equiv_{\Phi,s} a' \quad \text{iff} \quad a = a' \tag{2}$$

when $s \in V$, and

$$a \equiv_{\Phi,s} a' \quad \text{iff} \quad A_c(a) = A_c(a') \ \text{ for all } v \in V \text{ and all } c \in C_\Phi[z]_v \tag{3}$$

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 **behavioural equivalence** and denote it $\equiv$.

For $\Phi \subseteq \Sigma$, a **hidden Φ-congruence** on a hidden Σ-algebra A is a Φ-congruence $\simeq$ that is the identity on visible sorts, that is, 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**.

It is not hard to demonstrate the following fact.

Fact 24. Given a hidden signature Σ and a hidden subsignature Φ:

[†] However, real software development processes involve much more, including constantly evolving requirements and the resulting need to constantly evolve the software (Goguen 1994).

1 any hidden Φ-congruence is a hidden $(\Phi \cup \Psi)$-congruence;
2 $\Phi' \subseteq \Phi$ implies $\equiv_{\Phi} \subseteq \equiv_{\Phi'}$; and
3 behavioural Φ-equivalence is a hidden Φ-congruence.

However, the key property[†] can be stated as follows.

Theorem 25. If Σ is a hidden signature, Φ is a hidden subsignature of Σ, and A is a hidden Σ-algebra, then behavioural Φ-equivalence is the *largest* behavioural Φ-congruence on A.

Probably the most common case is $\Phi = \Sigma$, but the generalization to smaller Φ is useful, for example in verifying refinements, as we will see in Section 4.

Theorem 25 is not hard to prove: the proof generalizes the well-known construction of an abstract machine as a quotient of the term algebra by the behavioural equivalence relation, which is usually called the Nerode equivalence in that context (Meseguer and Goguen 1985; Malcolm 1996).

Theorem 25 implies that if $a \simeq a'$ under some hidden congruence $\simeq$, then a and a' are behaviourally equivalent. This justifies a variety of techniques for proving behavioural equivalence; see also Goguen and Malcolm (1994) and Malcolm and Goguen (1994). In this context, a relation may be called a **candidate relation** before it is proved to be a hidden congruence.

Example 26. Let A be any model of the FLAG theory in Example 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

$$A \models\!\!\!\equiv (\forall \texttt{F} : \texttt{Flag})\ \texttt{rev rev F} = \texttt{F}$$

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 behaviourally 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 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 (using the ordinary theorem of constants (Goguen and Malcolm 1996; Goguen 1998)). The following shows R is a hidden congruence:

[†] This elegant formulation appeared in a conversation between Grant Malcolm and Rolf Hennicker for the special case where $\Phi = \Sigma$.

```
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
```

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

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

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; alternatively, a decision procedure for the Booleans could be used instead of OBJ3's built-in Booleans, which only knows how to reduce ground terms. We believe the proof above is about as simple as could be hoped for. Actually, the third reduction above is unnecessary, but it is more trouble to justify its elimination than it is to ask OBJ to do the reduction. Section 3.2 discusses this example and the techniques that are needed.

The above is an example of what we call *hidden coinduction* – this will be explained further below. We now give some results to simplify hidden coinduction proofs. Suppose $\Sigma = \Gamma \cup \Delta$: the letters Γ and Δ are intended to suggest *generators* (also called *constructors*) and *destructors* (also called *selectors*), respectively (Malcolm and Goguen 1994). In Example 26, Δ contains up? and Γ contains all the other flag operations.

Corollary 27. If $\Sigma = \Delta \cup \Gamma$ and if $\equiv_\Delta$ on Σ-algebra A is preserved by Γ, then $\equiv_\Delta = \equiv_\Sigma$ on A. More generally, if $\Psi \subseteq \Phi = \Delta \cup \Gamma \subseteq \Sigma$ and $\equiv_\Delta$ is preserved by Γ, then $\equiv_\Delta = \equiv_\Phi$.

Proof. We show the more general result. By Fact 24, $\equiv_\Delta$ is a hidden $(\Delta \cup \Psi)$-congruence that contains behavioural Φ-equivalence, since $\Delta \subseteq \Phi$. If $\equiv_\Delta$ is preserved by Γ, then it is a hidden $\Delta \cup \Gamma = \Phi$-congruence, and the desired result follows from Theorem 25. □

Verifiers naturally want to do as little work as possible. Hence they do not want to bother with Ψ at all, and they do not want any overlap between Δ and Γ, that is, they want to use Δ and Γ such that $\Phi = \Delta + \Gamma + \Psi$, where '+' denotes disjoint union for operations and ordinary union for sorts. In the object paradigm, it is often natural to let Δ contain attributes and Γ methods. Then we can give a simple syntactic definition for $\equiv_\Delta$, as follows.

Proposition 28. If $\Phi = \Delta + \Gamma + \Psi$, where Δ consists of visible operations, if A is a Σ-algebra, and if we define $aR_\Delta a'$ iff $\delta(a, d) = \delta(a', d)$ for all $\delta \in \Delta$ and all $d \in A_w$ where w is the arity of δ, then $R_\Delta = \equiv_\Delta$. Thus, if R_Δ is preserved by Γ, then R_Δ is behavioural Φ-equivalence.

Proof. Part (3) of Definition 23 is equivalent to the definition of R_Δ because all operations in Δ are visible. □

The above shows that R as defined in Example 26 really is $\equiv_\Delta$. Furthermore, if Γ consists of methods and Δ of attributes, and if the equations satisfy a certain common property, then R_Δ is automatically preserved by Γ.

Definition 29. If $\Phi = \Delta + \Gamma + \Psi$, where operations in Δ are visible and in Γ are hidden, then a set E of Σ-equations is Δ/Γ-**complete** iff for all $\delta \in \Delta$, $m \in \Gamma$, there is some $t \in T_{\Delta\cup\Psi}(\{x\})$ such that using E we can prove

$$\delta(d, m(d', x)) = t \ ,$$

with x of hidden sort h', $\delta \in \Delta_{wh,v}$, $m \in \Gamma_{w'h',h}$, $d \in D_w$, and $d' \in D_{w'}$.

The following is a straightforward corollary to Proposition 28.

Proposition 30. If $\Phi = \Delta + \Gamma + \Psi$ with operations in Δ visible and in Γ hidden, if A is a hidden Σ-algebra, and if E is Δ/Γ-complete, then R_Δ is preserved by Γ, and therefore R_Δ is behavioural Φ-equivalence.

In the special case where all equations involving Γ have the form

$$\delta(m(x)) = t \ ,$$

with x of hidden sort h, $\delta \in \Delta$, $m \in \Gamma$, $t \in T_{\Delta\cup\Psi}(\{x\})$, it is easy to see that E is Δ/Γ-complete (this result was suggested to us by Răzvan Diaconescu).

To summarize, **hidden coinduction** is the proof technique where we define a relation, show it is a hidden congruence, and then show behavioural equivalence of two terms by showing they are congruent.

Exercise 31. Prove that the equation

```
    eq putx(M,putx(N,S)) = putx(M,S) .
```

is a behavioural consequence of the theory `X` in Example 8, and that it is not strictly satisfied. (A proof for the satisfaction part of this exercise appears in Appendix A.)

The way we define the congruence relation in a coinductive proof can have a significant effect on how the proof applies to models. If the relation is defined inductively over some constructors, then given a model A, the congruence is only defined on the subalgebra $A_0 \subseteq A$ generated by those constructors in A: this is the subalgebra that is reachable using those constructors. More specifically, the proof that such a candidate relation is a congruence might proceed by induction on the given constructors; in this case, what is proved is that the relation is a congruence on the subalgebra A_0. Usually we do not care whether or not a behavioural equation is satisfied by unreachable states, because these states cannot occur when the machine is run. An example of a correctness proof that applies only to reachable states is given in Example 35 in Section 4.

3.1. *A vending machine*

Since none of the examples of nondeterminism in Section 2.3 involve state, which is the most characteristic feature of hidden algebra, we really should give an example of nondeterminism with a hidden sort. For some reason, vending machines are very popular for illustrating nondeterminism and concurrency. The specification 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 (let us say) by `true` and `false`, respectively; and then it goes to a 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 behaviour by adding equations like

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

which says you cannot get the same substance three times in a row. It is interesting to look at the final algebra, which we will denote using F, for the signature without the constant `init`: according to our 'magic formula', it consists (up to isomorphism) of all infinite Boolean sequences – that is, it is the algebra of (what are called) *traces* in traditional concurrency theory. 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 behaviour of M. This simple and elegant situation holds in general for hidden algebraic models of nondeterministic concurrent systems.

3.2. *Derived operations*

A derived operation is one that can be defined in terms of other operations – intuitively, it can be eliminated; it is convenient but not necessary. However, derived operations can sometimes get in the way. For example, in showing $\simeq$ is a hidden Δ-congruence, we want to ignore the derived operations in Δ. This is justified by a result saying that given $\Delta \subseteq \Sigma$, if $\Delta' \subseteq \Delta$ consists of the non-derived operations in Δ, then $\simeq$ is a hidden Δ-congruence iff it is a hidden Δ'-congruence. This simplifies hidden congruence proofs.

The most difficult part of making this precise is to define what derived operations are. The easiest way to do this involves a little category theory.

Definition 32. Let (Δ, E) be a hidden theory and let $(\Delta', E') \subseteq (\Delta, E)$ be a subtheory. Let $\mathscr{A}$ and $\mathscr{A}'$ be the categories of models of (Δ, E) and (Δ', E'), respectively, and let $U : \mathscr{A} \to \mathscr{A}'$ be the forgetful functor. Then the operations in $\Delta - \Delta'$ are all **derived** iff there exists an inverse F to U (so that both are isomorphisms). In this case, we say that $(\Delta', E') \subseteq (\Delta, E)$ is a **deriving extension.** □

This long sought for general definition of derived operation applies beyond equational logics to any institution whose signatures are an inclusive category. We can now state the following generalization of the result informally sketched in the first paragraph of this subsection.

Proposition 33. Given a deriving extension $(\Delta', E') \subseteq (\Delta, E)$, a relation is a hidden Δ'-congruence iff it is a hidden Δ-congruence.

Proof. Note that the functors U and F do not change the underlying sets of models. Therefore, a relation R is a hidden Δ'-congruence iff the equivalence classes of R give a

(Δ', E')-model, iff (by applying F to this model) these equivalence classes give a (Δ, E)-model, that is, iff R is a hidden Δ-congruence. □

We will now illustrate this on the method `rev` of Example 26. If `rev` is derived, the third reduction in that example is unnecessary. Recall that the equation defining `rev` is

```
eq up? rev F = not up? F .
```

It is not obvious from this that `rev` is derived, so instead we use the following two conditional equations to define `rev`:

```
cq rev F = dn F if up? F .
cq rev F = up F if not up? F .
```

These two equations are behaviourally equivalent to the single equation, in the sense that the two theories define exactly the same model categories: it is not hard to see that a hidden algebra A satisfies the single equation iff it behaviourally satisfies the two conditional equations. Now define the functor $F : \mathscr{A}' \to \mathscr{A}$ to add to an $\mathscr{A}'$-model the unique operation `rev` defined by the above two equations. Now it is easy to see that this F is inverse to U, and hence is an isomorphism (and thus also a left adjoint). Thus, `rev` is derived. Eliminating one reduction from Example 3 is not worth the effort involved in this proof, but there certainly are other cases where such a proof would be worth the trouble, and the proof technique is of some interest in itself.

Note that in general there is no maximal deriving extension. It is quite possible for a signature to have many different subsignatures of derived operations. For example, we have just shown that `rev` can be defined in terms of `up` and `dn`, but we can also derive both `up` and `dn` from `rev` alone, using the following conditional equations:

```
cq up F = F     if up? F .
cq up F = rev F if not up? F .
cq dn F = rev F if up? F .
cq dn F = F     if not up? F .
```

4. Refinement

The simplest view of refinement assumes a specification (Σ, E) and an implementation A, and asks if $A \models\!\!\!\equiv_\Sigma E$. The generalization to behavioural satisfaction is significant here, as it allows us to treat many subtle implementation tricks that only 'act as if' correct, for example, 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 described in the introduction. To rise above this, we work with a specification E' for A, rather than an actual model†. This not only makes the proof far easier, but also has the

† Some may object that this manoeuver 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*. Empirical studies show that little of the difficulty of software development lies in the code itself (only about 5% (Boehm 1981)); much more of the difficulty lies in specification and design, and 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 Goguen and Malcolm (1996). Thus we have achieved a significant separation of concerns.

advantage that the proof will apply to any other model A' that satisfies E'. Hence, what we prove is $E' \models E$. In semantic terms, this means that any A satisfying E' also satisfies E, but very significantly, it also means that we can use hidden coinduction to do the proof. The remarks immediately preceding Section 3.1 about how an inductive definition of the congruence relation of a coinductive proof affects the models that it applies to are also relevant to coinductive refinement proofs.

Refinement is consistent with a view of software development as a series of design decisions giving a series of specifications that are progressively more refined and closer to actual code[‡]. In this view, the more abstract specifications allow more 'implementation freedom', while the more concrete specifications tend to have larger signatures and/or more defined terms. This is illustrated in the following.

Exercise 34. Show that each of `C1`, `C2`, `C3` below refines `A1`, and discuss how these refinements can be seen as reducing nondeterminism (*c.f.* the discussion in Section 2.3). The abstract specification `A1` says that objects have two natural-number-valued attributes, the first of which is less than the second:

```
th A1 is sort State .
  pr DATA .
  ops a b : State -> Nat .
  var S : State .
  eq a(S) < b(S) = true .
endth
```

The proofs that the following refine `A1` need a richer theory of `NAT` than that in Section 2.1, including a definition for addition and some simple lemmas about how it relates to inequality. We have concealed[†] these lemmas, so that our readers may have the pleasure of discovering them from the results of the reductions without the lemmas, as this is such an important and productive part of the verification process.

```
th C1 is sort State .
  pr DATA .
  ops a b : State -> Nat .
  var S : State .
  eq b(S) = s a(S) .
endth
```

The above satisfies the constraint in `A1` by letting the value of `b` be one more than that of `a`. In the next module, `b` can be any value greater than `a`:

```
th C2 is sort State .
  pr DATA .
  ops a b c : State -> Nat .
```

[‡] Empirical studies show that this view of software development is naive, since real development projects involve many false starts, redesigns, prototypes, patches, *etc.* (Button and Sharrock 1993). Nevertheless, the idealized view is still useful as a way to organize and document verification efforts, often retrospectively.

[†] They are in the source file for this paper, so that OBJ3 can read them to do the proofs, but they are invisible comments to LaTeX.

```
    var S : State .
    eq b(S) = a(S) + s c(S) .
  endth
```

In the following module, `b` must be at least three more than `a`:

```
  th C3 is sort State .
    pr DATA .
    ops a b c : State -> Nat .
    var S : State .
    eq b(S) = a(S) + c(S) .
    eq s s 0 < c(S) = true .
  endth
```

(*Hint*: Our proof uses a Skolem function.) Readers who want to try more proofs could determine which of the three concrete theories above are refinements of others.

4.1. *Stack as pointer plus array*

We now give a more substantive illustration of refinement, showing correctness of the familiar array-with-pointer implementation of a stack.

Example 35. We have to prove that the equations in the `STACK` specification of Example 19 are behavioural consequences of a specification for pairs of an array state and a pointer, enriched with the stack operations. Let Φ be the signature of `STACK` with E its equations, and let Σ be the signature of the implementation with E' its equations. $\Phi \subseteq \Sigma$, because the stack operations are defined in the implementation. We will show $E' \models_\Phi E$, which means that every appropriate model of the implementation gives rise to a model of `STACK` after forgetting the operations in Σ but not in Φ (the subscript Φ on $E' \models_\Phi E$ means that behavioural satisfaction of E will be in terms of Φ-contexts). Our proof uses hidden coinduction, without considering models at all, although our inductive definition of the candidate relation does affect the set of models to which it applies.

We represent the pointer by the state of a cell containing a natural number. A stack of depth n has n in this cell, and has its n elements in places $0, ..., n-1$ of the array. Instead of using the 'helper' results in Section 2.2, we apply Theorem 25 directly. The line '`pr ARR`' means that the models of `PTR||ARR` should include all and only the models of `ARR`, and, in particular, that they have the same data algebra as `ARR`, namely `DATA`.

```
  th PTR||ARR is sort Stack .
    pr ARR .
    op  _||_ : Nat Arr -> Stack [prec 10].
    op  empty : -> Stack .
    op  push : Nat Stack -> Stack .
    op  top_ : Stack -> Nat .
    op  pop_ : Stack -> Stack .
    var I N : Nat .  var A : Arr .
    eq  empty  =  0 || nil .
    eq  push(N, I || A)  = s I || put(N,I,A) .
    eq  top s I || A  =  A[I] .
    eq  top 0 || A  =  0 .
    eq  pop s I || A  =  I || A .
    eq  pop 0 || A  =  0 || A .
  endth

  th LEMMA is pr PTR||ARR .
    vars I I1 I2 : Nat .  vars A A1 A2 : Arr .
    cq I1 || put(I,I2,A) = I1 || A if not I2 < I1 .
  endth

  ***> relation + new constants used for quantifier elimination
  th R is pr PTR||ARR .
    op _R_ : Stack Stack -> Bool .
    op _R_ : Nat   Nat   -> Bool .
    vars I I1 I2 : Nat .  vars A A1 A2 : Arr .
    eq I1 R I2 = I1 == I2 .
    eq (0 || A1) R (I || A2) = I == 0 .
    eq (I || A) R (I || A) = true .
    eq (s I1 || A1) R (s I2 || A2) = I1 == I2 and A1[I1] == A2[I1] and
                                     (I1 || A1) R (I2 || A2) .
    ops n i j i1 i2 : -> Nat .
    ops a a1 a2   : -> Arr .
  endth

  ***> first show R a congruence using case analysis: i1=0 or i1=s(j):
  open R + LEMMA .
  eq i1 = 0 .
  ***> then expanding i1 || a1 R i2 || a2 gives
  eq i2 = 0 .
  ***> now check the congruence equations:
  red top(i1 || a1) R top(i2 || a2) .
  red pop(i1 || a1) R pop(i2 || a2) .
  red push(n, i1 || a1) R push(n, i2 || a2) .
  close

  open R + LEMMA .
  eq i1 = s j .
  ***> then expanding i1 || a1 R i2 || a2 gives the 3 equations below:
  eq i2 = s j .
  eq a2[j] = a1[j] .
  eq j || a1 R j || a2 = true .
  ***> now check the congruence equations:
  red top(i1 || a1) R top(i2 || a2) .
```

```
  red pop(i1 || a1) R pop(i2 || a2) .
  red push(n, i1 || a1) R push(n, i2 || a2) .
  close

  ***> finally check the stack equations:
  red pop empty R empty .
  red top push(n, i || a) R n .
  red pop push(n, i || a) R i || a .
```

All reductions give `true`. We believe this proof is about as simple as possible. Indeed, most of the text is specifications; the proof itself is just 27 lines (not counting `open`/`close` commands, variable declarations or comments, but including the proof of the lemma). OBJ3 does all the boring work in executing the 11 `red` commands, for a total of 120 rewrites. Formally, the congruence proofs use quantifier elimination, case analysis on `i1`, implication elimination, relation expansion, conjunction elimination, and finally reduction, where relation expansion makes explicit some consequences of `R` being true on a pair of terms.

Now we prove the lemma used above, which just says that values in the array that lie above the pointer do not matter:

```
 cq I || put(N,J,A) = I || A if not J < I ,
```

by proving

```
 cq I || put(N,J,A)  R  I || A = true if not J < I
```

with the following:

```
  open R .
  ***> base case:
    red 0 || put(n,j,a) R 0 || a .
  ***> induction step:
    eq not j < s i = true .
    eq i || put(n,j,a) R i || a = true .
    red s i || put(n,j,a) R s i || a .
  close
```

The reader can visit this proof on the world wide web, execute the OBJ proof score on a remote OBJ3 server, and follow links to explanation pages attached to proof pages, and to background information. The URL for the proofsite homepage is

```
      http://www.cse.ucsd.edu/groups/tatami/
```

Two points about this proof need further consideration. The first is that the lemma is used in proving that `R` is a congruence, and the proof of that lemma appears to rely upon `R` being a congruence – we treat this point in an exercise below, because the technique involved is useful for other problems. The second point is that the inductive definition of the congruence `R` implies that this proof only applies to states in models of the concrete specification that are reachable using that induction scheme (see the discussion just before Section 3.1). In particular, the proof that `R` is a congruence assumes that all stacks are of the form `I || A`. Therefore our proof only applies to states that are reachable by the `_||_` operation. Of course, we could add operations

```
    op getPointer : Stack -> Nat .
    op getArray   : Stack -> Array .
```

together with appropriate equations in order that all models of the concrete theory be reachable. Another possibility would be to use a different candidate relation, say

```
    S1 R S2  =  top S1 == top S2  and  (pop S1) R (pop S2) .
```

This relation would of course require a different correctness proof (Goguen and Malcolm 1994).

It is also worth noting that although this specification has two hidden sorts, we are mainly interested in one of them, namely `Stack`. Both sorts have visible-valued operations, but only the `Stack` sort's (single) visible-valued operation (namely `top`) is an attribute. Some subtle points regarding our use of OBJ3's built-in equality relation == are discussed in Section 1.1 of Goguen and Malcolm (1996).

Exercise 36. Show that the use of the lemma in the congruence proof is not circular, because its use there depends only on `R` being transitive and symmetric, not on its being a congruence. *Hint*: show that the proof of the lemma, together with transitivity and symmetry, justify the following:

```
    var I1 I2 I : Nat .  var A : Arr .  var S : Stack .
    cq I1 || put(I,I2,A) R S = S R I1 || A  if not I2 < I1 .
```

This proof was fairly straightforward to construct, except for the lemma. However, our style of using OBJ greatly facilitated even this, by producing an expression that suggested the lemma. This is typical of our experience with OBJ proof scores (Goguen 1990c; Goguen and Malcolm 1996; Goguen 1998).

Notice that in this implementation the term `top empty` is given the concrete value `0`, but it could have been given any other value, for example, by adding one of the equations

```
    eq top 0 || A = s 0 .
    eq top 0 || A = A[0] .
```

In fact, the above proof does not require any particular value to be specified; all that is necessary in order to prove that `R` is a congruence is that all empty stacks (*i.e.*, all stacks whose pointer is `0`) should give the same value for `top`. Thus, we might have added one of the equations

```
    eq top 0 || A = top 0 || nil .
    eq top 0 || A1 = top 0 || A2 .
```

although the latter is not a rewrite rule because of the free variable in the right-hand side. A fully satisfactory treatment of stacks requires order sorted algebra, and would also allow a more satisfactory proof of refinement, because it would allow us to disregard values of `top` and `pop` on empty stacks, so that the value of `top empty` could be left unspecified.

Exercise 37. Specify sets and lists and, by verifying an appropriate refinement, show that sets can be implemented with lists.

A more sophisticated view of refinement (Goguen and Meseguer 1982; Sannella and Tarlecki 1988; Hennicker 1990; Orejas *et al.* 1996) allows the concrete implementation to rename or even identify some of the abstract sorts and operations, thus giving rise to a hidden signature map from the abstract to the concrete signature.

Definition 38. A **hidden signature map** $\varphi : (H, \Sigma) \to (H', \Sigma')$ is a signature morphism $\varphi : \Sigma \to \Sigma'$ that preserves hidden sorts and is the identity on (V, Ψ). A hidden signature map $\varphi : \Sigma \to \Sigma'$ is a **refinement** $\varphi : (\Sigma, E) \to (\Sigma', E')$ iff $\varphi A' \models_\Sigma E$ for every (Σ', E')-algebra A'.

(In the above, $\varphi A'$ denotes A' viewed as a Σ-algebra.) It can be shown that φ is a refinement if all visible consequences of the abstract specification hold in the concrete specification (Malcolm and Goguen 1994).

Proposition 39. A hidden signature map $\varphi : (\Sigma, E) \to (\Sigma', E')$ is a refinement if $E' \models \varphi(c[e])$ for each $e \in E$ and each visible Σ-context c, where if e is the equation $(\forall X)\ t = t'$, then $c[e]$ denotes the equation $(\forall X)\ c[t] = c[t']$.

The following consequence of Corollary 27 justifies the use of hidden coinduction for proving correctness of refinements; some examples appear in Malcolm and Goguen (1994).

Proposition 40. A hidden signature map $\varphi : (\Sigma, E) \to (\Sigma', E')$ is a refinement if $\Sigma = \Gamma + \Delta + \Psi$, all operations in $\phi(\Gamma)$ preserve $\equiv_{\phi(\Delta)}$ for all (Σ', E')-algebras, and $E' \models \varphi(c[e])$ for each $e \in E$ and all visible Δ-contexts c.

We write $\phi(\Gamma)$ for the subsignature of Σ' whose operations are of the form $\phi(\gamma)$ for γ in Γ. Correctness proofs for refinements involve showing that the concrete specification has the desired behaviour, and generally make use of the concrete equations. The proposition above says that a refinement can be proved correct using the concrete equations to verify both the congruence property of $\equiv_{\phi(\Delta)}$ and the satisfaction of the equations $\varphi(c[e])$.

4.2. *Model-based refinement*

Early studies of refinement were model oriented (Hoare 1972), considering refinement a relationship between two models, one ‘abstract’ and the other ‘concrete’. Then it makes sense to map from concrete variables to the abstract objects they represent. However, the (often) complex representations of the concrete program, and the (usually) complex semantics of the programming language in which it is expressed introduce gratuitous difficulties into such proofs. Our approach to refinement simplifies the first difficulty by considering *theories* for both the concrete and the abstract levels, while the complexity of the programming language semantics becomes a completely separate issue. In particular, our more abstract definition of refinement for specifications allows stepwise refinement to begin before choosing concrete representations for variables; such a choice corresponds to fixing a model, and it is good engineering practice to delay such a commitment as long as possible.

For us, a representation is correct if it is a model of the concrete theory, and showing this should be much easier than showing that it satisfies the abstract specification, because the representations will be much closer. The perhaps initially mysterious fact that mappings go in opposite directions for specifications and for models is explained at a higher level of abstraction by the theory of institutions (Goguen and Burstall 1992), which shows that in logics satisfying certain mild assumptions, it is natural that the maps induced by a signature morphism on models and on theories should go in opposite directions. Hence, the duality between model-based and theory-based refinement is very natural.

Finally, note that hidden algebra allows subtle changes of representation to be proved correct much more easily; indeed, our primary motivation is always to make correctness proofs just as easy as possible.

5. Concurrent connection

Concurrency is an essential part of the object paradigm, and is natural to hidden algebra, in that no order of execution is specified by hidden theories – in particular, concurrent execution is legal whenever it is possible. This section describes an elegant construction of composite systems from components using the *concurrent connection* (a weaker version of this construction was called the *independent sum* when first introduced in Goguen and Diaconescu (1994b)). As motivation, consider the following example.

Example 41. Recall the specification `X` of an integer cell X in Example 8, and define another integer cell, Y, by everywhere replacing `X` and `x` with `Y` and `y`, respectively. Then the concurrent connection of X and Y should have a specification $X \parallel Y$ with a single hidden sort, where the operations of X and Y have the same semantics as before, such that operations in X and Y do not interfere with each other. Thus $X \parallel Y$ should be the union of the specifications X and Y, with their sorts identified, and with some new equations to express the noninterference of X and Y, as follows:

```
th XY is sort State .
  pr DATA .
  ops putx puty : Nat State -> State .
  ops (getx_) (gety_) : State ->  Nat .
  var S : State .
  vars M N : Nat .
  eq getx putx(N,S) = N .
  eq gety puty(N,S) = N .
  eq getx puty(N,S) = getx S .
  eq gety putx(N,S) = gety S .
endth
```

The last two equations express the noninterference of X and Y.

Exercise 42. Prove that the equation

```
  eq putx(M,puty(N,S)) = puty(N,putx(M,S)) .
```

is a behavioural consequence of theory `XY` in Example 41. (A solution appears in Appendix A.)

We now give a generalization and universal characterization of the above construction to any pair of specifications (it extends to any set of specifications).

Definition 43. A **synchronization** of specifications P_1, P_2 with just one hidden sort is a specification P with refinements $\varphi_1 : P \to P_1, \varphi_2 : P \to P_2$, where P is called the **shared part**, and a **connection** of a synchronization φ_1, φ_2 is a specification Q with refinements $\psi_i : P_i \to Q$ such that

- $Q \models (\forall X)(\forall Y)(\forall S)\ \psi_i(a_i)(X, \psi_j(m_j)(Y, S)) = \psi_i(a_i)(X, S)$,

- $Q \models (\forall X)(\forall Y)(\forall S)\ \psi_i(m_i)(X, \psi_j(m_j)(Y, S)) = \psi_j(m_j)(Y, \psi_i(m_i)(X, S))$, and
- $\varphi_1 ; \psi_1 = \varphi_2 ; \psi_2$,

for $i = 1, 2$, where S is an h-sorted variable, m_k is a method and a_k is an attribute of P_k, such that none of these symbols lie in the image of φ_k. We call an initial connection the **concurrent connection**, if there is one.

In practice, we can avoid these commutativity equations by representing states as ordered tuples, as in Example 35. This is possible because we do not mix messages into the same 'soup' as objects. Intuitively, the concurrent connection is 'the best' among all specifications with noninterfering insertions for the P_i. Although the components P_i must have just one hidden sort, the candidate connections Q need not have this property. However, the proof of the result below (slightly generalizing the proof in Goguen and Diaconescu (1994b) shows that it does have just one hidden sort.

Theorem 44. Every synchronization $\varphi_1 : P \to P_1, \varphi_2 : P \to P_2$ has a concurrent connection.

Proof. The concurrent connection Q has the signature obtained by combining the signatures of P_1 and P_2, identifying those parts that come from the shared part P. This gives signature morphisms $\psi_i : \Sigma_{P_i} \to \Sigma_Q$ with $\varphi_1 ; \psi_1 = \varphi_2 ; \psi_2$. The equations of Q are $\psi_i(\overline{E_1}) \cup \psi_2(\overline{E_2}) \cup I_\varphi$, where I_φ represents the noninterference axioms of Definition 43, and where we recall that

$$\overline{E_i} = \{c[e] \mid e \in E_i, c \in C_{\Sigma_i}[z]_v\} .$$

It is straightforward to check that this is an initial connection (Cîrstea 1996). □

Thus the specification XY of Example 41 is the concurrent connection of X and Y synchronized over the sort State, where the shared part P might (for example) be

```
th STATE is sort State .
  pr DATA .
endth
```

where DATA contains at least NAT and the Booleans.

The proof of the theorem above shows that in general, concurrent connections have an infinite number of equations. However, in many cases a finite number suffices – for example, if $\Sigma = \Delta + \Gamma + \Psi$, where Δ consists of visible operations, and $\equiv_\Delta$ is preserved by Γ, as in Proposition 30.

We can prove that a concurrent system is free from deadlock by proving that it is consistent. This is because deadlock means that the equations expressing synchronization do not have a solution.

Hidden coinduction can be used to verify properties of systems built by concurrent connection, as in the concurrent connection of an array and a pointer implementing a stack in Example 35. Communication protocols provide many further examples where correctness proofs are desirable. For example, we might have an abstract specification of a persistent buffer that ignores incoming values when it is full. Its specification would include an equation like

```
  cq in(S,V) = S  if full(S) .
```

where `in` is the method that sends an input to the buffer. This kind of buffer can be implemented by the concurrent connection of a buffer that always accepts incoming values (possibly overwriting values if it is full) and a 'gatekeeper' object that only allows incoming values when the buffer is not full – hidden coinduction gives an elegant correctness proof. Some examples of hidden correctness proofs for asynchronous communication protocols are given in Veglioni (1997).

The concurrent connection of two objects without synchronization is their coproduct in the category having appropriate specifications as objects and certain refinements as morphisms. More generally, Corina Cîrstea has shown that concurrent connection with synchronization is colimit in this category (Cîrstea 1996).

Finally, we emphasize that the definition of concurrent connection in this section is not really suitable for proofs, but instead provides an abstract characterization of the intended semantics. For proving behavioural properties, the ordered tuple approach used in Example 35 is much better, because it avoids the extra complication of the commutativity equations.

6. Conclusions and related work

The hidden approach described in this paper uses behavioural satisfaction to get an algebraic treatment of state that abstracts away from implementation details. The idea of behavioural satisfaction was introduced by Reichel (Reichel 1981) in the context of partial algebras (see also Reichel (1985)). Behavioural equivalence of states, a generalization of bisimulation, appeared in Goguen and Meseguer (1982). Reichel's notion of behavioural theory has been developed further in several different directions within the algebraic specification community, mainly using partial algebras, for example, see Ehrig *et al.* (1983), Ehrig *et al.* (1993) and Bidoit (1996), and the survey Orejas *et al.* (1996). The first effective algebraic proof technique for behavioural properties was context induction, introduced by Hennicker (Hennicker 1990) and further developed with Bidoit (Bidoit 1996). This research programme is similar to ours in many ways, though their approach is more concerned with semantics than with proofs, and their context induction can be very awkward to apply in practice (see Gaudel and Privara (1991) for a discussion of some of the difficulties). We propose hidden coinduction as a way to eliminate the awkwardness of context induction.

Reichel's seminal work on behavioural satisfaction was in part motivated by an insight into how to unify initial and final semantics (Reichel 1981). Behavioural and final semantics were perhaps first advocated by Montanari *et al.* (Giarrantana *et al.* 1976), though Wand (Wand 1979) also made an early contribution. Finality is also used for treating states in Reichel (1981), Goguen and Meseguer (1982), Meseguer and Goguen (1985), and Malcolm (1996), among many other places, including the present paper – there is some elegant more recent by work by Reichel on co-algebraic semantics for the object paradigm (Reichel 1995). This flood of work on finality and behavioural abstraction validates some intuitions expressed long ago by Guttag (Guttag 1975; Guttag 1977).

There is also a distinguished tradition of research in *coalgebra*. One thread in this tradition seeks to show existence of final transition systems, which give rise to an abstract notion of bisimulation and can be used to give a semantics for process algebras (Aczel and Mendler 1989; Barr 1993). Another thread views coalgebra as a variation on universal algebra (Rutten 1996), and applies it to functional programming (Hagino 1988; Malcolm 1990; Gordon 1995), to automata theory (Rutten and Turi 1994; Rutten 1996), and to the object paradigm (Reichel 1995; Jacobs 1995; Jacobs 1996; Jacobs 1997; Cîrstea 1996). An interesting recent development combines algebra and coalgebra to describe denotational and operational semantics (Turi and Plotkin 1997).

Reichel (Reichel 1995) was the first to apply coalgebra explicitly to the object paradigm, and his basic construction can be used to show that hidden algebra extends coalgebra with generalised constants (Malcolm 1996; Cîrstea 1997). It is precisely this extension that allows the treatment of nondeterminism we advocate in this paper. In fact, it seems difficult to treat nondeterminism at all in a purely coalgebraic approach, since the obvious move of using power objects in the defining functor compromises the effectiveness of equational reasoning. Our hidden coinduction is a consequence of the fact that behavioural equivalence is the largest hidden congruence, and our use of the term 'coinduction' agrees with its use in Milner and Tofte (1991), which seems to be the earliest use of the term for a proof principle. However, coinduction does arise very naturally in a coalgebraic setting (Jacobs 1995; Malcolm 1996). We have not set out here in great detail how standard constructions from the object paradigm are modelled in hidden algebra, but Goguen and Diaconescu (1994b) and Cîrstea (1997) describe a hidden approach to inheritance. These are also treated in a coalgebraic setting in, for example, Jacobs (1996).

We are experimenting with ways to organize hidden proofs as active websites, using HTML, Java, JavaScript, *etc*., and a website editor called Kumo (Goguen *et al.* 1997a; Goguen *et al.* 1997b), which provides direct support for hidden coinduction and automatically generates an entire website for a proof, including executable OBJ3 proof scores, links to background material, and to explanation pages. We intend to link this tool to decision methods for special domains beyond canonical term rewriting theories, such as Presburger arithmetic. Traceability is very important when constructing complex new proofs, and we intend to explore use of the TOOR hypermedia tool (Pinheiro and Goguen 1996) for this purpose. We have already done one rather substantial hidden proof, namely the correctness of an optimizing compiler for OBJ3, based on an abstract term rewriting machine (Hamel and Goguen 1994; Hamel 1996), and several smaller examples are on the web, including the stack example of Section 4.

This paper has restricted attention to hidden many sorted algebra. The extension to hidden order sorted algebra is not really difficult, but it cannot be trivial, since it covers nonterminating systems, partial recursive functions, multiple inheritance, error definition and handling, coercion, overwriting, multiple representation, and more – many details appear in Malcolm and Goguen (1994), but there is still more work to be done. We also wish to further explore connections with other approaches, including coalgebra and concurrent logic programming. For example, it would be interesting to find morphisms between the relevant institutions, generalizing the adjunctions of Winskel (Winskell 1984).

We feel that hidden algebra is a natural next step in the evolution of algebraic specification, carrying forward the intentions of its founders in a simple and elegant way to the realities of modern software. Initial algebra semantics still works for data values, but now we can also handle systems of objects (abstract machines), concurrency, constraints, streams, existential queries, and more. We wish to explore this potent combination of paradigms further, and apply it to further problems of real practical value.

Appendix A. Two small coinductive proofs

Below are OBJ3 proofs using hidden coinduction to solve two of the exercises in this paper.

Exercise 31: We let Δ contain `getx` and Γ contain `putx`. Then we use Proposition 28 and apply Corollary 27.

```
***> to prove putx(M,putx(N,S)) = putx(M,S):
openr X .
  op _R_ : State State -> Bool .
  var S1 S2 : State .
  eq S1 R S2 = getx S1 == getx S2 .
  ops s1 s2 : -> State .
  ops n  m  : -> Nat .
close

***> first show R is a congruence:
open .
  eq getx s1 = getx s2 .
  red putx(n,s1) R putx(n,s2) .  ***> should be: true
close
***> now prove the equation:
red putx(m,putx(n,s1)) R putx(m,s1) .   ***> should be: true
```

Exercise 42: Let Δ contain `getx` and `gety`, and Γ contain `putx` and `puty`. Then use Proposition 28 and apply Corollary 27.

```
***> to prove putx(M,puty(N,S)) = puty(N,putx(M,S)):
openr XY .
  op _R_ : State State -> Bool .
  var S1 S2 : State .
  eq S1 R S2 = getx S1 == getx S2 and gety S1 == gety S2 .
  ops s1 s2 : -> State .
  ops n  m  : -> Nat .
close

***> first prove R is a congruence:
open .
  eq getx s1 = getx s2 .
  eq gety s1 = gety s2 .
  red putx(n,s1) R putx(n,s2) .  ***> should be: true
  red puty(n,s1) R puty(n,s2) .  ***> should be: true
```

```
close
***> now prove the equation:
red putx(m,puty(n,s1)) R puty(n,putx(m,s1)) .   ***> should be: true
```

References

Aczel, P. and Mendler, N. (1989) A final coalgebra theorem. In: Pitt, D. H. *et al.* (ed.) Category Theory and Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **389**.

Barr, M. (1993) Terminal coalgebras in well-founded set theory. *Theoretical Computer Science* **114** 299–315.

Bidoit, M., Hennicker, R. and Wirsing, M. (1996) Behavioural and abstractor specifications. *Science of Computer Programming* **25** (2–3).

Boehm, B. (1981) *Software Engineering Economics*, Prentice-Hall.

Button, G. and Sharrock, W. (1993) Occasioned practises in the work of implementing development methodologies. In: Jirotka, M. and Goguen, J. (eds.) *Requirements Engineering: Social and Technical Issues*, Academic Press 217–240.

Cîrstea, C. (1996) *A Semantic Study of the Object Paradigm*, Transfer thesis, Programming Research Group, Oxford University.

Cîrstea, C. (1997) Coalgebra semantics for hidden algebra: parameterized objects and inheritance. Paper presented at the 12th Workshop on Algebraic Development Techniques, June 1997.

Diaconescu, R. (1994) *Category-based Semantics for Equational and Constraint Logic Programming*. Ph. D. thesis, Programming Research Group, Oxford University.

Diaconescu, R. (1996) A category-based equational logic semantics to constraint programming. In: Haveraaen, M., Owe, O. and Dahl, O.-J. (eds.) Recent Trends in Data Type Specification. *Springer-Verlag Lecture Notes in Computer Science* 200–222.

Ehrig, H., Kreowski, H.-J., Mahr, B. and Padawitz, P. (1983) Algebraic implementation of abstract data types. *Theoretical Computer Science* **20** 209–263.

Ehrig, H., Orejas, F., Cornelius, F. and Baldamus, M. (1993) Abstract and behaviour module specifications. Technical Report 93–25, Technische Universität Berlin.

Gaudel, M.-C. and Privara, I. (1991) Context induction: an exercise. Technical Report 687, LRI, Université de Paris-Sud.

Giarrantana, V., Gimona, F. and Montanari, U. (1976) Observability concepts in abstract data specifications. Proceedings, Conference on Mathematical Foundations of Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **45**.

Goguen, J. (1990a) Higher-order functions considered unnecessary for higher-order programming. In: Turner, D. (ed.) *Research Topics in Functional Programming*. University of Texas at Austin Year of Programming Series, Addison Wesley 309–352.

Goguen, J. (1990c) Proving and rewriting. In: Kirchner, H. and Wechler, W. (eds.) Proceedings, Second International Conference on Algebraic and Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **463** 1–24.

Goguen, J. (1994) Requirements engineering as the reconciliation of social and technical issues. In: Jirotka, M. and Goguen, J. (eds.) *Requirements Engineering: Social and Technical Issues*, Academic Press 165–200.

Goguen, J. (1998) *Theorem Proving and Algebra*, MIT Press (to appear).

Goguen, J. and Burstall, R. (1992) Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery* **39** (1) 95–146.

Goguen, J. and Diaconescu, R. (1994a) An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science* **4** 363–392.

Goguen, J. and Diaconescu, R. (1994b) Towards an algebraic semantics for the object paradigm. In: Ehrig, H. and Orejas, F. (eds.) Proceedings, Tenth Workshop on Abstract Data Types. *Springer-Verlag Lecture Notes in Computer Science* **785** 1–29.

Goguen, and Malcolm, G. (1994) Proof of correctness of object representation. In: Roscoe, A. W. (ed.) *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice Hall 119–142.

Goguen, J. and Malcolm, G. (1996) *Algebraic Semantics of Imperative Programs*, MIT Press.

Goguen, J., Malcolm, G. and Kemp, T. (1998) A hidden Herbrand theorem. In: Palamidessi, C., Glaser, H. and Meinke, K. (eds.) Principles of Declarative Programming. *Springer-Verlag Lecture Notes in Computer Science* **1490** 445–462.

Goguen, J. and Meseguer, J. (1982) Universal realization, persistent interconnection and implementation of abstract modules. In: Nielsen, M. and Schmidt, E. M. (eds.) Proceedings, 9th International Conference on Automata, Languages and Programming. *Springer-Verlag Lecture Notes in Computer Science* **140** 265–281.

Goguen, J. and Meseguer, J. (1987a) Order-sorted algebra solves the constructor selector, multiple representation and coercion problems. In: *Proceedings, Second Symposium on Logic in Computer Science*, IEEE Computer Society 18–29.

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

Goguen, J., Mori, A. and Lin, K. (1997a) Algebraic semiotics, proof Webs, and distributed cooperative proving. In: *Proceedings, User Interfaces for Theorem Provers* (Sophia Antipolis, France, 1–2 Sept. 1997) 25–34.

Goguen, J., Mori, A., Lin, K., Roşu, G. and Sato, A. (1997b) Distributed cooperative formal methods tools. In: *Proceedings, Automated Software Engineering* (Lake Tahoe NV, 3–5 Nov 1997), IEEE 55–62.

Goguen, J., Thatcher, J. and Wagner, E. (1978) An initial algebra approach to the specification, correctness and implementation of abstract data types. In: Yeh, R. (ed.) *Current Trends in Programming Methodology, IV*, Prentice Hall 80–149.

Gordon, A. D. (1995) Bisimilarity as a theory of functional programming. *Electronic Notes in Theoretical Computer Science* **1**.

Guttag, J. (1975) *The Specification and Application to Programming of Abstract Data Types*, Ph. D. thesis, University of Toronto. (Computer Science Department, Report CSRG–59.)

Guttag, J. (1977) Abstract data types and the development of data structures. *Communications of the Association for Computing Machinery* **20** 297–404.

Hagino, T. (1988) A typed lambda calculus with categorical type constructors. In: Pitt, D. H., Poigné, A. and Rydeheard, D. E. (eds.) Category Theory and Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **283** 140–157.

Hamel, L. (1996) *Behavioural Verification and Implementation of an Optimizing Compiler for OBJ3*, Ph. D. thesis, Oxford University Computing Lab, 1996.

Hamel, L. and Goguen, J. (1994) Towards a provably correct compiler for OBJ3. In: Hermenegildo, M. and Penjam, J. (eds.) Proceedings, Conference on Programming Language Implementation and Logic Programming. *Springer-Verlag Lecture Notes in Computer Science* **844** 132–146.

Hennicker, R. (1990) Context induction: a proof principle for behavioural abstractions. In: Miola, A. (ed.) Proceedings, International Symposium on the Design and Implementation of Symbolic Computation Systems. *Springer-Verlag Lecture Notes in Computer Science* **429** 101–110.

Hoare, C. A. R. (1972) Proof of correctness of data representation. *Acta Informatica* **1** 271–281.

Jacobs, B. (1995) Mongruences and cofree coalgebras. In: Nivat, M. (ed.) Algebraic Methodology and Software Technology (AMAST95). *Springer-Verlag Lecture Notes in Computer Science* **936** 245–260.

Jacobs, B. (1996) Objects and classes, coalgebraically. In: Freitag, B., Jones, C., Lengauer, C. and Schek, H.-J. (eds.) *Object-Orientation with Parallelism and Persistence*, Kluwer 83–103.

Jacobs, B. (1997) Invariants, bisimulations and the correctness of coalgebraic refinements. Technical Report CSI–R9704, Computer Science Institute, University of Nijmegen.

Malcolm, G. (1990) Data structures and program transformation. *Science of Computer Programming* **14**.

Malcolm, G. (1996) Behavioural equivalence, bisimilarity, and minimal realisation. In Haveraaen, M., Owe, O. and Dahl, O.-J. (eds.) Recent Trends in Data Type Specifications. *Springer-Verlag Lecture Notes in Computer Science* **389**.

Malcolm, G. and Goguen, J. (1994) Proving correctness of refinement and implementation. Technical Monograph PRG-114, Programming Research Group, University of Oxford.

Meseguer, J. and Goguen, J. (1985) Initiality, induction and computability. In: Nivat, M. and Reynolds, J. (eds.) *Algebraic Methods in Semantics*, Cambridge University Press 459–541.

Milner, R. and Tofte, M. (1991) Co-induction in relational semantics. *Theoretical Computer Science* **87** (1) 209–220.

Orejas, F., Navarro, M. and Sánchez, A. (1996) Algebraic implementation of abstract data types: a survey of concepts and new compositionality results. *Mathematical Structures in Computer Science* **6** (1).

Pinheiro, F. and Goguen, J. (1996) An object-oriented tool for tracing requirements. *IEEE Software* (Special issue of papers from ICRE '96) 52–64.

Reichel, H. (1981) Behavioural equivalence – a unifying concept for initial and final specifications. In: *Proceedings, Third Hungarian Computer Science Conference*, Akademiai Kiado, Budapest.

Reichel, H. (1985) Behavioural validity of conditional equations in abstract data types. In: *Contributions to General Algebra 3*, Teubner (Proceedings of the Vienna Conference, June 21–24).

Reichel, H. (1995) An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science* **5** 129–152.

Roşu, G. and Goguen, J. (1998) Hidden congruent deduction. To appear in *Proceedings, International Workshop on First Order Theorem Proving*.

Rutten, J. J. M. J. (1996) Universal coalgebra: a theory of systems. Technical Report CS–R9652, CWI.

Rutten, J. and Turi, D. (1994) Initial algebra and final coalgebra semantics for concurrency. In: de Bakker, J., de Roever, J. W. and Rozenberg, G. (eds.) Proc. REX Symposium 'A Decade of Concurrency'. *Springer-Verlag Lecture Notes in Computer Science* **803** 530–582.

Sannella, D. and Tarlecki, A. (1988) Toward formal development of programs from algebraic specifications. *Acta Informatica* **25** 233–281.

Turi, D. and Plotkin, G. (1997) Towards a mathematical operational semantics. In: Proceedings Logic in Computer Science 1997.

Veglioni, S. (1997) *Integrating Static and Dynamic Aspects in the Specification of Open, Object-based and Distributed Systems*, Ph. D. thesis, Oxford University Computing Laboratory.

Wand, M. (1979) Final algebra semantics and data type extension. *Journal of Computer and System Sciences* **19** 27–44.

Winskel, G. (1984) Categories of models for concurrency. In: Brooks, S., Roscoe, A. W. and Winskel, G. (eds.) Proceedings, Workshop on the Semantics of Concurrency. *Springer-Verlag Lecture Notes in Computer Science* **197** 246–267.