# Formalising Ontologies and Their Relations

Trevor Bench-Capon and Grant Malcolm

Department of Computer Science, University of Liverpool
PO Box 147, Liverpool L69 3BX, UK

**Abstract.** Ontologies allow the abstract conceptualisation of domains, but a given domain can be conceptualised through many different ontologies, which can be problematic when ontologies are used to support knowledge sharing. We present a formal account of ontologies that is intended to support knowledge sharing through precise characterisations of relationships such as compatibility and refinement. We take an algebraic approach, in which ontologies are presented as logical theories. This allows us to characterise relations between ontologies as relations between their classes of models. A major result is *cocompleteness* of specifications, which supports merging of ontologies across shared sub-ontologies.

## 1 Introduction

Over the last decade ontologies — best characterised as explicit specifications of a conceptualisation of a domain [17] — have become increasingly important in the design and development of knowledge based systems, and for knowledge representations generally. They have been shown to be useful in

- knowledge acquisition (e.g. [23]);
- knowledge reuse (e.g. [19]) and sharing (e.g. [17]); and
- verification and validation of knowledge based systems (e.g. [3]).

It has become recognised that even in a given domain multiple conceptualisations, and hence ontologies, are possible. The conceptualisations will inevitably be oriented to the tasks to be carried out, and are likely to be influenced also by personal tastes, and may even reflect fundamental disagreements as to the nature of the domain. But though they differ, ontologies will be related to one another. Thus knowledge reuse requires that ontologies be refined and extended to tailor them to some new application; sharing knowledge between systems requires that the individual system ontologies be compatible and even, at least in part, common, or capable of being systematically related to some third ontology common to the two systems.

Thus the programme of ontologies requires that we be able to speak of ontologies being consistent, compatible, the result of merging two other ontologies, an extension of some ontology, or a refinement of some ontology. We may even wish to speak of some ontology being closer to a given ontology than some third ontology, to facilitate knowledge sharing, or to inform the choice of an ontology to refine when designing a new knowledge based system. Unfortunately, although intuitions abound, precise characterisations of these relations do not.

We believe that these notions need to be made precise. Therefore in this paper we will set out an abstract and formal way of describing ontologies, so that we can give precise characterisations of these relations in these terms. Accordingly Sect. 2 sets out our formalism for describing ontologies, and Sect. 3 illustrates its use in characterising these relations with some motivating examples.

## 2 Ontological Specifications

Semantics is of primary importance in designing any language, and a language for formally specifying ontologies should have a clear and precise semantics. In particular, relationships between ontologies are best described in terms of their semantics. We take an algebraic approach, in which ontologies are presented as algebraic theories. This allows us to characterise relations between ontologies as relations between their classes of models. In this section we present an abstract syntax for specifying ontologies and develop the model theory that provides semantics for these specifications. We are much less interested in presenting a concrete syntax; the motivating examples given below are in an *ad hoc* syntax.

For reasons of space, the presentation below assumes some familiarity with algebraic specifications (introductions can be found in [22, 9]) and also with basic category theory (see [20, 1] for introductions). A more detailed presentation, including proofs absent from the present paper, can be found in [2].

Our ontologies specify classes of entities with attributes. These attributes take values in data types such as numbers, booleans, lists and so on. Sometimes it is convenient to make changes in the types of attributes, for example an ontology may be refined by refining the types of some of its attributes, so we consider these types to be a part of ontological specifications. We formalise this part of specifications using the notion of *order-sorted algebraic theory*; [14, 11] give details of order-sorted algebra, the following is an example of a theory for the natural numbers, in the notation of the language OBJ [16, 11].

```
th NAT is
  sorts Nat NonZeroNat .
  subsort NonZeroNat < Nat .
  op 0 : -> Nat .
  op s : Nat -> NonZeroNat .
  op p : NonZeroNat -> Nat .
  var N : Nat .
  eq  p(s(N)) = N .
endth
```

The details of this notation are not essential to the present paper. The main point is that an order-sorted theory presents some sorts, partially ordered by a subsort relation, some typed operations, and some axioms (here just one equation).

Models of such specifications interpret sorts as sets, and operations as functions; these sets and functions should satisfy the axioms in the obvious sense.

The data types in our ontologies are given by an order-sorted theory together with a fixed model interpreting its sorts and operations:

**Definition 1.** *A* **data domain** *is a pair $(T, D)$, where $T$ is an order-sorted equational theory $T = (\mathsf{S}, \Sigma, E)$, where $\mathsf{S} = (S, \leq)$ is the partially ordered set of sort names, $\Sigma$ is the collection of typed operation symbols, and $E$ is the set of equations, and where $D$ is a model of $T$. We often write $D$ instead of $(T, D)$.*

We do not require that $D$ be an initial model of $T$ [15], although in many cases that would be an obvious choice. Any computable algebra can be specified equationally [4]; even uncomputable algebras, such as the reals, can be used in ontological specifications by fixing an appropriate model $D$ (for example, the reals provide a model of the theory `NAT`). An important advantage of using order-sorted algebra is that it is implemented in languages such as OBJ [16, 11] and CafeOBJ [8]. This provides support for prototyping and theorem proving.

A *theory morphism*[1] $\phi : T \to T'$ between order-sorted theories induces a functor from models of $T'$ to models of $T$ (see, e.g., [10], though Proposition 7 presents the essential ideas). This means that any model $D'$ of $T'$ gives rise to a model $\phi D'$ of $T$; this allows us to define morphisms of data domains as follows:

**Definition 2.** *A* **morphism of data domains** *$\psi : (T, D) \to (T', D')$ is a pair $\psi = (\phi, h)$, where $\phi = (f, g) : T \to T'$ is an order-sorted theory morphism and $h : \phi D' \to D$ is a $T$-homomorphism; We require that if $\sigma'$ in $\Sigma'$ returns values of sort $f(s)$ for $s \in S$, then $\sigma' = g(\sigma)$ for some $\sigma$ in $\Sigma$.*

Data morphisms go from coarse to fine structures. For example, one ontology might specify an attribute with values in a data type $\mathsf{shade}$, interpreted in one domain as $\{\mathsf{light}, \mathsf{dark}\}$, while another ontology may specify values of this attribute in the range $[0, 100)$. A domain morphism might translate values up to 50 as being $\mathsf{light}$, and values between 50 and 100 as being $\mathsf{dark}$. The mapping

$$i \mapsto \begin{cases} \mathsf{light} & \text{if } 0 \leq i < 50 \\ \mathsf{dark} & \text{if } 50 \leq i < 100 \end{cases}$$

describes the homomorphism part of the above definition (i.e., we have $h_{\mathsf{shade}}$ mapping $i$ in $\phi D'_{\mathsf{shade}} = (0, 100]$ to $D_{\mathsf{shade}} = \{\mathsf{light}, \mathsf{dark}\}$).

**Definition 3.** *An* **ontology signature***, or just* **signature** *for short, is a triple $(\mathcal{D}, \mathsf{C}, A)$, where $\mathcal{D} = (T, D)$ is a data domain, $\mathsf{C} = (C, \leq)$ is a partial order, called a* **class hierarchy***, and $A$ is a family of sets $A_{c,e}$ of* **attribute symbols** *for $c \in C$ and $e \in C + S$, where $S$ is the set of sorts in the order-sorted theory $T$. This family of sets is such that $A_{c',e} \subseteq A_{c,e'}$ whenever $c \leq c'$ in $\mathsf{C}$ and $e \leq e'$ in $\mathsf{S} + \mathsf{C}$.*

[1] Order-sorted theory morphisms (see [14]) are pairs $(f, g)$, where $f$ is a monotonic map on sort names, and $g$ maps operation symbols to operation symbols.

For $c \in C$ and $s \in S$, the set $A_{c,s}$ represents attributes of the class $c$ that take values in $D_s$. Attributes may also take values in classes: for $c, c' \in C$, the set $A_{c,c'}$ represents attributes of the class $c$ that take values in the class $c'$. The final condition of the above definition states that if $c \leq c'$ then the class $c$ inherits all attributes of the class $c'$.

*Example 4.* We give an ontology for different kinds of cars that might be used by a second-hand car dealer, using an *ad hoc* notation that hopefully requires little explanation. There are four classes, `Car`, `Estate-Car`, `Saloon-Car` and `Model`. Each class is followed by an indented list of its attributes and their result types. We omit the data domain from the specification, but names beginning with a `*` represent data types, and `{white,red,blue}` represents a data type with elements `white`, `red`, and `blue`.

```
Car
  colour: {white,red,blue}
  model: Model
  year:  *year
  price: *pounds

Estate-Car < Car
  rear-space:  *square-metres

Saloon-Car < Car
  hatchback: {y, n}

Model
  name: *string
  manufacturer: {maker1, maker2}
  photo: *gif
```

The class `Car` has four attributes, e.g., `model`, which takes values in the class `Model` (i.e., `model` $\in A_{\texttt{Car},\texttt{Model}}$). The class `Estate-Car` has five attributes: four inherited from `Car` plus the attribute `rear-space`, which takes values in a data type `*square-meters` (we omit the data domain from this specification, but it would be sensible for this data type to consist of some kind of numerical values).

We turn now to the semantics of such specifications:

**Definition 5.** *A* **model** $M$ *of an ontology signature* $(\mathcal{D}, \mathsf{C}, A)$ *consists of:*

- *A* $\mathsf{C}$*-sorted family of sets* $M_c$ *for* $c \in C$*, called the* **carriers** *of* $M$*;*
- *for each attribute symbol* $\alpha \in A_{c,e}$*, a function* $M_{c:\alpha} : M_c \to M_e$*, where if* $e \in S$ *then* $M_e = D_e$*, subject to the following monotonicity requirement: for* $c \leq c' \in C$ *and* $\alpha \in A_{c',e}$*, we have* $M_{c':\alpha}\restriction_{M_c} = M_{c:\alpha}$*.*

*Because of the above monotonicity property, we write simply* $M_\alpha$ *instead of* $M_{c:\alpha}$*.*

*Given models* $M$ *and* $N$ *of* $(\mathsf{C}, A)$*, a* **model homomorphism** $h : M \to N$ *is a* $\mathsf{C}$*-sorted family of functions* $h_c : M_c \to N_c$ *such that for all* $\alpha \in A_{c,e}$*,* $h_c; N_\alpha = M_\alpha; h_e$*, where* $h_e = 1_{D_e}$ *if* $e \in S$*.*

An ontology consists of a signature and some axioms; its denotation is the class of all models that satisfy the axioms. Before we give the formal definitions, we consider morphisms that relate ontologies:

**Definition 6.** *An* **ontology signature morphism** $\chi : (\mathcal{D}, \mathsf{C}, A) \to (\mathcal{D}', \mathsf{C}', A')$ *is a tuple $\chi = (\phi, f, g)$, where $\phi : \mathcal{D} \to \mathcal{D}'$ is a data domain morphism, $f : \mathsf{C} \to \mathsf{C}'$ is a morphism of partial orders, and $g$ is a family of functions $g_{c,e} : A_{c,e} \to {A'}_{f(c),f(e)}$, where $f(e) = \phi(e)$ if $e \in S$ (i.e., if $e \in S$ then $g_{c,e}$ maps attributes that take values in $D_e$ to attributes that take values in $(\phi D')_e$), such that if $c \leq c'$ and $e \leq e'$ then $g_{c',e}(\alpha) = g_{c,e'}(\alpha)$ for all $\alpha \in A_{c',e}$.*

Signature morphisms give rise to functors on models:

**Proposition 7.** *A signature morphism $\chi : (\mathcal{D}, \mathsf{C}, A) \to (\mathcal{D}', \mathsf{C}', A')$ induces a functor, which we also call $\chi$, from the category of $(\mathcal{D}', \mathsf{C}', A')$-models to the category of $(\mathcal{D}, \mathsf{C}, A)$-models. The functor is defined on objects by*

- $(\chi M)_c = M_{f(c)}$ *for $c \in C$, and*
- $(\chi M)_\alpha = M_{g_{c,e}(\alpha)}$ *for $\alpha \in A_{c,e}$,*

*where $M$ is a model of $(\mathcal{D}', \mathsf{C}', A')$.*

The final component in an ontology consists of axioms constraining the possible values of attributes. Here, we consider axioms to be just conditional equations, but other kinds of sentences, such as horn clauses, are also possible.

**Definition 8.** *An* **ontology** *is a pair $(\Sigma, Ax)$ where $\Sigma$ is a signature and $Ax$ a set of axioms. A* **model** *of such an ontology is a model $M$ of $\Sigma$ that satisfies each axiom in $Ax$, in which case we write $M \models Ax$.*

*An* **ontology morphism** $\chi : (\Sigma, Ax) \to (\Sigma', Ax')$ *is a signature morphism $\chi : \Sigma \to \Sigma'$ such that $\chi M \models Ax$ whenever $M \models Ax'$.*

*Example 9.* Here is another car ontology that might be used for a buyers' guide:

```
Car
  colour: {white, red, blue}
  model: Model

Model
 name: *string
 type: {estate, saloon}
 hatchback: {y, n, n/a}
 picture: *gif
 rear-space: *square-metres
 manufacturer: Motor-maker
 price: *pounds

Motor-Maker
  name: {maker1, maker2}
  dealer: Dealer
```

```
Dealer
  name: *string
  address: *string

Axioms:
  var M : Model
  hatchback(M) = n/a  if  type(M) = estate
  rear-space(M) = 0   if  type(M) = saloon
```

The axioms in this ontology give specific values of attributes `hatchback` and `rear-space` depending on the `type` of model.

## 3 Relating Ontologies

Ontology morphisms provide a direct way of relating ontologies; any morphism $\chi : \mathcal{O} \to \mathcal{O}'$ states that the ontology $\mathcal{O}'$ is a *refinement* of the ontology $\mathcal{O}$. This is because any model of $\mathcal{O}'$ gives rise to, by Proposition 7 and Definition 8, a model of $\mathcal{O}$. This approach is standard in algebraic specification (see, e.g., [24]).

A more general way of relating ontologies is through pairs of morphisms:

**Definition 10.** *A* **relation** *between ontologies $\mathcal{O}_1$ and $\mathcal{O}_2$ consists of an ontology $\mathcal{O}$ and a pair of morphisms $\chi_i : \mathcal{O} \to \mathcal{O}_i$ for $i = 1, 2$.*

A morphism $\chi : \mathcal{O}_1 \to \mathcal{O}_2$ is a special case of a relation where $\mathcal{O} = \mathcal{O}_1$.

### 3.1 Combining and Sharing Ontologies

Ontologies represent knowledge of a given domain from a specific point of view. Examples 4 and 9 specify models of cars, from the points of view of car dealers and buyers' guides, respectively. In order to support knowledge sharing, it is important that ontologies can be merged, as stated in the following theorem:

**Theorem 11.** *The category of ontologies and their morphisms is cocomplete.*

This is stated in the language of category theory [20], and says that there is a 'best' way (a *colimit*) of combining related ontologies that share some subcomponents. A proof is given in [2]; an example is given below. Colimits are standard in combining algebraic theories (for example [5]; see also [18], whose cocompleteness results for order-sorted theories are used in the proof of Theorem 11).

*Example 12.* The following is a common subcomponent for the ontologies of Examples 4 and 9.

```
Car                             Model
  colour: {white,red,blue}        name: *string
  model: Model                    manufacturer: {maker1, maker2}
                                  photo: *gif
```

There is an inclusion from this ontology (call it $\mathcal{O}$) to the ontology (call it $\mathcal{O}_1$) of Example 4. This gives us $\chi_1 : \mathcal{O} \rightarrow \mathcal{O}_1$. To relate $\mathcal{O}_1$ to the ontology (call it $\mathcal{O}_2$) of Example 9, we construct a morphism $\chi_2 : \mathcal{O} \rightarrow \mathcal{O}_2$ as follows. The class `Car` and its attributes are included in $\mathcal{O}_2$. The class `Model` and its attribute `name` are likewise included, and the attribute `photo` is mapped to `picture` in $\mathcal{O}_2$. The attribute `manufacturer` is mapped to the *compound* attribute `manufacturer;name` in $\mathcal{O}_2$, i.e., take the `manufacturer` attribute, giving a value in the class `Motor-Maker`, then take the attribute `name` of that class (this means that this example is not a colimit in the sense of Theorem 11; see [2]).

In the colimit there is just one attribute `photo` representing both `photo` from $\mathcal{O}_1$ and `picture` from $\mathcal{O}_2$. These two attributes are the image of `photo` in $\mathcal{O}$ under $\chi_1$ and $\chi_2$ respectively, so that this attribute is shared between the two ontologies, just as all the attributes of the class `Car` are shared. However, there are two `manufacturer` attributes in the colimit; one representing the attribute of the same name from $\mathcal{O}_1$, and the other (which we call `manufacturer2`) representing the attribute from $\mathcal{O}_2$. The morphisms $\chi_i$ state that `manufacturer` and `manufacturer2;name` should be the same, so in the colimit there is an axiom that constrains all models to treat them in the same way; this is the first axiom in the colimit of our two related ontologies, which is given in full below:

```
Car
  colour: {white, red, blue}
  model: Model
  year:  *year
  price: *pounds

Estate-Car < Car
  rear-space:  *square-metres

Saloon-Car < Car
  hatchback: {y, n}

Model
 name: *string
 type: {estate, saloon}
 hatchback: {y, n, n/a}
 picture: *gif
 rear-space: *square-metres
 manufacturer: {maker1, maker2}
 manufacturer2: Motor-maker
 price: *pounds

Motor-Maker
  name: {maker1, maker2}
  dealer: Dealer

Dealer
  name: *string
  address: *string
```

```
Axioms:
  var M : Model
  manufacturer(M) = name(manufacturer2(M))
  hatchback(M) = n/a  if  type(M) = estate
  rear-space(M) = 0   if  type(M) = saloon
```

Note that ontologies can be related in different ways, and the ontology $\mathcal{O}$ does not really correspond to the 'intersection' of two ontologies, but is created in such a way as to state just what is to be shared between ontologies. For example, we might have omitted the `manufacturer` attribute from $\mathcal{O}$, which would correspond to a judgement that the two attributes of that name in $\mathcal{O}_1$ and $\mathcal{O}_2$ were distinct. We might have included an attribute `price` in $\mathcal{O}$, mapped to the attribute of the same name in $\mathcal{O}_1$ and to the compound attribute `model;price` in $\mathcal{O}_2$. This would express the judgement that these two attributes correspond to the same feature. However, in this example, we have not done this because these features are in fact distinct; for the second-hand car dealer, the price is an individual car's saleable value, and for a buyers' guide, the price is the model's new price.

### 3.2 Compatibility of Ontologies

Our definition of model requires data domains to be interpreted in a fixed way. This means that not all ontologies have models, because the axioms in an ontology might not be satisfiable. That is, the axioms might be inconsistent in the sense that they entail *true* = *false*, while the fixed data model in the data domain requires that these values be distinct. We say that an ontology is **consistent** iff it has at least one model (so that the axioms are satisfiable).

**Definition 13.** *Let $\chi_i : \mathcal{O} \to \mathcal{O}_i$ for $i = 1, 2$ be a relation between ontologies. We say that $\mathcal{O}_1$ and $\mathcal{O}_2$ are* **compatible (over** $\mathcal{O}$**)** *iff their colimit is consistent.*

**Proposition 14.** *Let $\chi_i : \mathcal{O} \to \mathcal{O}_i$ for $i = 1, 2$ and let $M_i$ be a model of $\mathcal{O}_i$ for $i = 1, 2$ such that $\chi_1 M_1 = \chi_2 M_2$; then the colimit of $\chi_i$ has a model, and so $\mathcal{O}_1$ and $\mathcal{O}_2$ are compatible over $\mathcal{O}$.*

A proof is given in [2]. Intuitively, this says that if models of $\mathcal{O}_1$ and $\mathcal{O}_2$ agree on their shared parts (i.e., on $\mathcal{O}$), then those models can be 'merged' to provide a model of the colimiting ontology. Similar results in an algebraic setting can be found in [21, 7].

## 4 Conclusions

We have presented an algebraic approach to formalising ontologies, and used this to capture various relationships between ontologies. Ontologies are specified as classes partially ordered by inheritance, and these classes have attributes that take values either in data types or in other classes. Axioms constrain the values that these attributes can take. The semantics of ontologies is given by classes of models, and this allows us to give precise characterisations of relationships between ontologies at the abstract level of relationships between their classes of models. Refinement corresponds to the existence of a morphism from an 'abstract' ontology to a more 'concrete' one, merging is achieved through colimits, and compatibility refers to the consistency of colimiting ontologies.

The advantages of an algebraic approach include a clean and precise semantics, and the existence of standard tools and techniques that can be used in proving properties such as refinement of ontologies. The main cocompleteness result (Theorem 11), draws on a long tradition of combining algebraic theories through colimits. This also allows us a certain amount of freedom in the logical language used for axioms. In this paper, we opt for equational axioms, because these are simple yet expressive; the study of algebraic theories at the level of institutions [10] shows that many other logics, such as Horn clause logic, are possible, while still retaining cocompleteness.

Our ontologies are similar to 'hidden algebra' [13, 12] in having a fixed data universe (hidden algebra also has a notion of behavioural equivalence that is not relevant to our present purposes). This allows us to reuse some results from hidden algebra. For example, because of the fixed interpretation of data domains, we know that standard equational deduction is sound but not complete for ontologies.

We have concentrated in this paper on developing an algebraic framework for specifying ontologies, rather than giving a concrete syntax for specifications (beyond that used in the examples). The results in Sections 2 and 3 represent a good start to our programme of characterising relations between ontologies, but there is more work to be done. Larger examples would provide validation of the definitions of the various relationships between ontologies, and more importantly suggest new kinds of relationships useful in practice. For instance, the brief examples in this paper use very simple relations between the class hierarchies, and we envisage that real knowledge sharing through ontologies will use rather more sophisticated relationships. It would also be interesting to relate our approach to other possible formalisms for ontologies, such as feature structures [6], which we feel will also prove interesting in relating ontologies to database schemata.

## References


[1] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[2] Trevor Bench-Capon and Grant Malcolm. Relating ontologies. Draft available at `http://www.csc.liv.ac.uk/~grant/ps/`, 1999.

[3] Trevor Bench-Capon. The role of ontologies in the verification and validation of knowledge based systems. In R. Wagner, editor, *Proceedings of the 9th International Workshop on Database and Expert Systems*, pages 64–69. IEEE Press, Los Alamitos, 1998.

[4] Jan A. Bergstra and John V. Tucker. Algebraic specifications of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.

[5] Rod Burstall and Joseph A. Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on*

*Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86.

[6] Robert L. Carpenter. *The Logic of Typed Feature Structures*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge, 1992.

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

[8] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[9] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.

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

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

[12] Joseph A. Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 1999. To appear.

[13] Joseph A. Goguen and Grant Malcolm. Hidden coinduction: behavioral correctness proofs for objects. *Mathematical Structures in Computer Science*, 1999. To appear.

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

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

[16] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph A. Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Practice*. to appear. Also available as a technical report from SRI International.

[17] Thomas R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*. Kluwer Academic Publishers, 1993.

[18] Anne E. Haxthausen and Friederike Nickl. Pushouts of order-sorted algebraic specifications. In *AMAST '96*. Springer-Verlag Lecture Notes in Computer Science 1101, 1996.

[19] Gertjan van Heijst, Guus Schreiber, and Bob J. Wielinga. Using explicit ontologies in KBS development. *International Journal of Human Computer Interaction*, 45:183–192, 1997.

[20] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, 1971.

[21] Grant Malcolm. Interconnection of object specifications. In Stephen Goldsack and Stuart Kent, editors, *Formal Methods and Object Technology*. Springer Workshops in Computing, 1996.

[22] Karl Meinke and John V. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, pages 189–411. Oxford University Press, 1993.

[23] Mark A. Musen. *Automated Generation of Model-Based Knowledge-Acquisition Tools*. Research Notes in Artificial Intelligence. Pitman, 1989.

[24] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications. *Acta Informatica*, 25:233–281, 1988.