CLASSIC: A Structural Data Model for Objects

Alexander Borgida
Ronald J. Brachman
Deborah L. McGuinness
Lori Alperin Resnick

AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

CLASSIC is a data model that encourages the description of objects not only in terms of their relations to other known objects, but in terms of a level of intensional structure as well. The CLASSIC language of *structured descriptions* permits i) partial descriptions of individuals, under an 'open world' assumption, ii) answers to queries either as extensional lists of values or as descriptions that necessarily hold of all possible answers, and iii) an easily extensible schema, which can be accessed uniformly with the data. One of the strengths of the approach is that the same language plays multiple roles in the processes of defining and populating the DB, as well as querying and answering.

CLASSIC (for which we have a prototype main-memory implementation) can actively discover new information about objects from several sources: it can recognize new classes under which an object falls based on a description of the object, it can propagate some deductive consequences of DB updates, it has simple procedural recognizers, and it supports a limited form of forward-chaining rules to derive new conclusions about known objects.

The kind of language of descriptions and queries presented here provides a new arena for the search for languages that are more expressive than conventional DBMS languages, but for which query processing is still tractable. This space of languages differs from the subsets of predicate calculus hitherto explored by deductive databases.

1 Motivation

A database is normally used to maintain a model of some aspect of reality. Traditional data models, such as the relational one, have achieved great efficiency in data storage and retrieval by restricting modeling power; in particular, the database is assumed to be a complete and accurate model of the world, where all the individual objects are restricted to be primitive values like numbers and strings, and all their inter-relationships are known and expressly stated. While undeniably of extensive value, this makes traditional data models unsuitable for a number of situations, for example,

- when *complex objects* are the natural way of describing the domain;
- when information about the domain is *incomplete* or becomes available *incrementally*;
- when the database should be taking a more *active* role in deducing relationships rather than being just a passive repository of data.

These situations include those in which new artifacts are being designed (e.g., CAD/CAM, configuration), or an understanding of some existing situation is being built up over time (e.g., diagnostic situations).

The field of *logic* (or *deductive*) *databases* [14] has emerged as one response to some of these weaknesses: incomplete information can be expressed naturally in logical languages using disjunction and existential quantifiers, and the database can infer new relationships through deductive rules. The chief drawback of this approach is computational intractability: a general version of this problem is equivalent to theorem proving for first order logic, and hence known to be undecidable. There has been progress towards finding various kinds of deductive rules [6] that can be added to relational systems without making the result hopelessly inefficient. But, as shown by results in [17], the same is not true for databases with partial knowledge: very simple kinds of disjunction and existentials cause intractability. The CLASSIC data model strives to provide a limited—and more tractable—set of deductions derived not from syntactic alterations to a mathematically-oriented language of predicates and quantifiers, but from a class of languages explicitly built for modeling real-world concepts and objects. As we shall see, CLASSIC also provides a means of dealing effectively with some types of partial knowledge.

A second motivation for our work is that research in both standard and deductive databases usually assumes that the query must return an *extensional answer*—a list of atomic individuals (or tuples thereof) that satisfy the relationship expressed by the query. However, once we assume that the database may not be complete, but might have both partial knowledge and active rules, it seems perfectly reasonable to allow *descriptive answers*—descriptions of objects, even when their properties are not fully known in the database, and descriptions of the necessary properties of all objects that will satisfy the query, even if some of those objects are not yet known. Some previous work in logic databases [11, 18] has considered descriptive or intensional answers, but

*With Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

as sufficient rather than necessary conditions for the answer set.

We should emphasize that although our goals appear ambitious, we are committed to a view of database management systems as having *limited* responsibilities, which can be carried out efficiently. Such a system may need to be used in conjunction with other computing engines (e.g., programming languages, rule bases) in order to solve general problems, like troubleshooting, configuration, etc. We do not want to put the burden on the database—even a deductive one—of having to support the entire range of activity in a complex problem-solving task. We take the database component of a complex application to be a cache for persistent information of limited complexity.

2 The CLASSIC approach: structured concepts

A CLASSIC database is mostly a repository of information about *individual objects*, or *individuals*, for short. As usual in object-based approaches [13, 16], objects have an intrinsic identity, and are related to each other through binary relationships; these are called *roles* in our case (elsewhere known as attributes or properties). Individuals will be grouped into collections indirectly by way of descriptions that apply to all members of a collection—these descriptions we shall call *concepts* (elsewhere known as classes).

All of this is standard. Where CLASSIC differs is in the ways we propose to address the problems presented in the previous section: rather than starting from the syntax of first-order logic, we begin with a *language for describing the general nature and structure of objects*—a language of structured concepts. This language will be used in several ways:

- to define the schema—an extended vocabulary of identifiers used in descriptions;
- to enter possibly incomplete information about individuals into the database;
- to express queries;
- to provide intensional descriptions of answers;
- to allow the specification of simple 'triggers'—limited forward-chaining inference rules.

2.1 Features of the CLASSIC language

The CLASSIC "data definition language" is a language of expressions for *defining concepts in a compositional manner.* Some concepts are obtained by grouping together individuals extensionally; others group individuals implicitly, through the use of intensional descriptions concerning their structure. Complex CLASSIC concepts are formed by composing expressions using a small set of constructors. We have chosen a prefix-style notation for our concept constructors, and each legitimate CLASSIC expression has one of these constructors as its first element.

2.1.1 Extensional constructors

The simplest kind of description one can form in CLASSIC is a *primitive concept.* Primitive concepts are simple, but not necessarily atomic; each primitive concept except for the topmost concept (which we call THING) is expected to have at least one parent (more general) concept. The simplest kind of primitive is one whose only parent is essentially vacuous, namely THING. For example, the concept of a CAR might be defined in this way:

(PRIMITIVE THING car),

where 'car' here is just an atomic index, and does not automatically represent the name of the concept (e.g., it could just as easily have been an integer, etc.).[1] This expression means that whatever it designates is simply a type of THING, with some unspecified difference from THING in general. Since the *differentia* is unspecified, one cannot definitively decide membership in a primitive concept simply by looking at the parts of an object. This is quite the opposite of the case with other, non-primitive concepts, as we shall see in a moment. Primitive concepts with the same parent but with different indices are distinct.

Note that the above could well have been written syntactically as

class CAR is-a THING

in some semantic data model, but we have chosen our notation because our concepts will be composite. Expressions can take full concept expressions in many places as subparts, and this notation supports embedded expressions quite naturally.

Primitives can also have non-trivial parents. Thus, SPORTS-CAR might be defined as a subconcept of both CAR and another concept, EXPENSIVE-THING:[2]

(PRIMITIVE (AND CAR EXPENSIVE-THING)
sports-car).

In fact, the parent of a primitive concept can be any CLASSIC concept, including another primitive. Primitives thus specify *necessary conditions*: *if* Corvette-1 is an instance of SPORTS-CAR, *then* it is both a CAR and an EXPENSIVE-THING. But note that there is no *sufficiency condition* specified for primitive concepts.

Another simple way of defining a concept is as nothing more than a time-invariant set of objects, specified by enumeration. *Enumerated concepts* have a fixed extent; for example, AMERICAN-CAR-MAKER might be defined as

(ONE-OF GM Ford Chrysler),

where GM, Ford and Chrysler are individual objects.

Primitive concepts and enumerations allow the definition of the kinds of class hierarchies that are found in conventional semantic data models.

2.1.2 Restriction-based constructors

The CLASSIC language of concepts allows us to go substantially beyond the simple IS-A hierarchies of more traditional semantic data models. It offers three special ways of describing objects in terms of their *structure*. As we shall see, these constructors give the language substantially more power, and in particular, let some class membership relations be determined by inference.

CLASSIC's three complex constructors are *role value restrictions*, cardinality *bounds*, and *co-reference constraints*.

Role value restrictions are type constraints that hold of the fillers for some single role. For example, the concept expression

(ALL thing-driven CAR)

describes any object that is related by the thing-driven role solely to individuals describable by the concept CAR (i.e., something "all of whose things-driven are CARs").

[1] In this paper we use the following orthographic conventions: CLASSIC symbols in all upper case are CONCEPTS; those in all lower case are roles or indices; and those in mixed case are Individuals. Concept-forming constructors, like **PRIMITIVE**, are written in **THIS FONT**.

[2] The **AND** constructor will be described momentarily in Section 2.1.3.

Bounds restrict the number of fillers for roles. For example,

```
(AT-MOST 4 thing-driven)
```

describes any object that is related to at most 4 distinct individuals through the **thing-driven** role.

```
(AT-LEAST 3 wheel)
```

describes any object that is related to at least 3 distinct individuals through the **wheel** role.

Co-reference constraints specify simple equalities between single-valued roles or, more generally, chains of such roles. For example, the expression

```
(SAME-AS (driver)
         (insurance payer))
```

describes all those individuals whose filler for the **driver** role is the same as the **payer** of their **insurance** role. This constraint is part of the *meaning* of any concept in which it appears, and is not just an integrity constraint. In fact, each of the constructors presented in this section can act as part of both necessary and sufficient conditions for concepts in which they appear (as long as they are not used in a primitive concept, in which case there are no sufficient conditions).

2.1.3 Composition

The CLASSIC language of concept expressions is fully compositional, with conjunction—the constructor **AND**—providing the 'glue'. For example,

```
(AND STUDENT
     (ALL thing-driven
          (AND SPORTS-CAR
               (ALL maker ITALIAN-COMPANY)))
     (AT-LEAST 1 thing-driven)
     (AT-MOST 2 thing-driven))
```

refers to objects that are instances of **STUDENT** and have the **thing-driven** role filled by one or two instances of **SPORTS-CAR**, each of which must have an **ITALIAN-COMPANY** as its **maker**. **AND** takes any number of concept expressions and forms a description that is the conjunction of those descriptions.

2.1.4 Tests

There are a multitude of other concept constructors that we might have considered adding to the CLASSIC language. But the result would have been both baroque and very likely intractable from a computational point of view. For practical purposes, we have chosen to add just one feature that can act as a limited 'steam valve'—*test concepts.* A test concept has an associated unary function in the host implementation language (e.g., LISP or C), which must return a boolean value; the concept then denotes all those objects for which the test function returns true. For example, the concept **EVEN-INTEGER** could be defined as

```
(TEST even),
```

where *even* is a host-language-specific procedure of one argument that returns true if and only if its argument is an even number. If the concept of an **INTEGER** were already defined (such a concept is built-in to the LISP implementation of CLASSIC), then this concept would instead be

```
(AND INTEGER
     (TEST even)).
```

Originally intended as a single simple facility for defining such 'concepts' as integer ranges, limited-precision numbers, limited-length strings, etc., the **TEST** facility has proven to be a pragmatically useful 'escape hatch' in tackling realistic applications (for example, a computer configuration task we have recently undertaken, with a CLASSIC database representing the parts inventory). While the construct is open-ended, it has been homogeneously integrated into the CLASSIC language. It can be thought of as a dual construct to **PRIMITIVE**; instead of allowing the specification of a primitive necessary condition (which **PRIMITIVE** gives us), it allows specification of a primitive sufficiency condition. Of course, a construct like **TEST** can undermine the tractability of inference in CLASSIC if used unwisely; if this is deemed unacceptable, one can consider **TEST** to lie outside of the core CLASSIC data model, and to be merely an implementation convenience. As we find patterns of use of this construct that can be integrated with the core data model in a semantically clean and tractable way, we expect to extend the core to include them, and again keep a firm lid on tractability in the core of CLASSIC.

In Appendix A, we summarize the syntax of CLASSIC concept (and individual) expressions. The particular concept constructors used in CLASSIC were chosen for their descriptive adequacy and minimality. For example, we did not define the constructor **EXACTLY-ONE**, which is easily derivable as the **AND** of **AT-LEAST 1** and **AT-MOST 1**. It is our intention to add a macro-definition facility in order to allow syntactic extensions such as **EXACTLY-ONE**, which might simplify CLASSIC expressions. Also, the usefulness of the particular choices of constructors here is supported in part by the history of knowledge representation languages [9] used in Artificial Intelligence. A great deal of work over the last ten years on "frame" representation systems has led to some consensus on the core constructs that are indispensable in representing object-oriented knowledge.

2.2 Implied relationships between concepts

It is important to note that, unlike in many semantic data models, the meaning of concepts in CLASSIC is determined by their structure—the composition of the particular constructors used to build them. This implies that certain relationships exist between concepts by virtue of their definition. For example, it is quite possible for several different concept expressions to denote the same class:

```
(AND (ALL thing-driven CAR)
     (ALL thing-driven EXPENSIVE-THING))
```

is the same concept as

```
(ALL thing-driven
     (AND CAR EXPENSIVE-THING)),
```

while

```
(ALL thing-driven
     (AND (ONE-OF Ford-1 Volvo-2 Toyota-3)
          (ONE-OF Volvo-2 Toyota-3 VW-4)))
```

is equivalent to

```
(AND (ALL thing-driven
          (ONE-OF Volvo-2 Toyota-3))
     (AT-MOST 2 thing-driven)).
```

Note that individuals may appear as part of the definition of concepts. Since **ONE-OF** was originally meant to introduce the equivalent of enumerated types in Pascal, and since definitions are not supposed to change meaning over time, inferences concerning the equivalence of concepts are affected only by the *identity* of such individuals, not by their properties, structure, etc.

While there is no room to include it here, we have a formal account of the meaning of the CLASSIC language, providing a denotational semantics for concepts. Concept meanings are functions that map database states to the sets of objects that 'satisfy' the conceptual descriptions in that state. Based on this semantics, two concepts are equivalent if and only if their denotations are identical functions. Of course, in practice, we use this specification to guide an implementation that manipulates the structures in an effective, procedural manner in order to decide equivalences. The recognition of all the necessary equivalences is the kind of inference that is at the core of the limited deduction and query processing performed by the CLASSIC system.

As an aside, note that the **TEST** constructor is an entirely procedural 'black box', so that complete reasoning cannot be done about the relationships between such concepts. In this way, **TEST** concepts act just like primitive ones. Admittedly, users may miss this subtle point, but our experience indicates that the inclusion of this constructor has been quite successful. Also, we see a certain resemblance between admitting "external" agents such as tests in CLASSIC, and the adoption of abstract data types as domains for the formerly pure relational data model.

We shall consider next the various ways of interacting with a CLASSIC database, and hence the ways of using the language of concepts.

3 Interacting with CLASSIC

The general pattern of using a database is to *define a schema*, using a data definition language (DDL); then successively (or concurrently) present *updates*, presented in a data manipulation language (DML) (which are either accepted or rejected because of constraint violations), and *queries*, in a query language (which are answered in some answer language). We shall consider the equivalent aspects of interacting with a CLASSIC database, and do so by describing a number of operators that can be applied to it.

3.1 Defining the schema

In a CLASSIC database, schema definition consists of giving names to various concepts, roles and individuals that appear of interest to all users, thus establishing a shorthand vocabulary for interacting with the database. Specifically, this is done through the use of operators such as **define-role**[3] and **define-concept**:

```
define-concept[AMERICAN-CAR-MAKER,
               (ONE-OF GM Ford Chrysler)];
define-role[thing-driven];
define-concept[RICH-KID,
          (AND STUDENT
               (ALL thing-driven SPORTS-CAR)
               (AT-LEAST 2 thing-driven))].
```

This last definition, for example, assigns to "RICH-KID" the concept of a student that drives at least two things, all of which are sports cars.

In contrast with standard databases, this process can be interleaved with updates and queries, so that we can define a new concept any time it seems useful. In lieu of a data dictionary, CLASSIC offers operators that allow concepts to be inspected; these will be described in Section 3.5.

3.2 Updating the database

CLASSIC individuals are created, and possibly named,[4] by the **create-ind** operator; for example,

```
create-ind[Rocky]
```

creates an individual named "Rocky", about whom nothing is known (except that Rocky is a THING). This establishes a unique identity for this individual, independent of its properties. Thereafter, information about an individual is added incrementally by the operator **assert-ind**, which allows three kinds of facts to be asserted about an individual in the form of expressions. The first two deal with the relationship of individuals to other individuals:

- the individual is related to some other individual by an (existing) role; so, for example, to add Volvo-17 to the group of things driven by Rocky, we would use

  ```
  assert-ind[Rocky,
             (FILLS thing-driven Volvo-17)];
  ```

- it is asserted that there are no additional objects related by the particular role to this individual, beyond those already known; thus,

  ```
  assert-ind[Rocky, (CLOSE thing-driven)]
  ```

  "closes" the thing-driven role so that no further fillers can be added.

The last expression points out an important design decision for CLASSIC databases: since we are committed to supporting incomplete information about objects, we do *not* make the 'closed-world' assumption [26] that a relationship does not hold unless we know of it. Thus, unless an individual's role is closed, there may be more objects related to it by that role, and so, for example, we would not yet have an upper bound on the number of role fillers for that role of the individual.

The third kind of fact that we can assert about an individual is that it is an instance of some concept—that is, the individual is appropriately described by the concept. In the case of primitive concepts, this is just the usual way of asserting class membership:

```
assert-ind[Rocky, PERSON].
```

Even in this case, one can already have a form of indeterminacy: if PERSON has other primitive subconcepts (e.g., INTELLECTUAL), it is left open at the moment whether Rocky is or is not describable by any of them. The concept constructors allow further forms of indeterminacy, as indicated by the following examples:

-
  ```
  assert-ind[Rocky,
             (AT-LEAST 1 thing-driven)]
  ```

  asserts the existence of an object driven by Rocky without explicitly naming it;

-
  ```
  assert-ind[Rocky,
             (ALL thing-driven SPORTS-CAR)]
  ```

  asserts that the objects driven by Rocky are SPORTS-CARs without knowing what they are, or how many there are;

-
  ```
  assert-ind[Rocky,
         (ALL thing-driven
              (ALL maker (ONE-OF Ferrari)))]
  ```

  requires that the objects driven by Rocky have maker Ferrari, without identifying those objects or any other properties they might have.

[3] **define-role** simply makes the database aware that a particular identifier will be used as a role name, thus allowing the DBMS to later detect errors such as typos. In some related systems [9], roles have more structure than they do here.

[4] If the database is large enough, it may be undesirable to require the user to name all individuals explicitly.

One can also directly assert membership of an individual in a concept constructed with the **AND** operator. Such an assertion amounts to a shorthand—it could be expanded into a series of **assert-ind** calls, one for each conjunct from the definition of the composed concept. For example, if RICH-KID is defined as above,

```
assert-ind[Rocky, RICH-KID]
```

would be the equivalent of

```
assert-ind[Rocky, STUDENT]
assert-ind[Rocky,
           (ALL thing-driven SPORTS-CAR)]
assert-ind[Rocky, (AT-LEAST 2 thing-driven)].
```

Since CLASSIC honors the semantics of constructors like **AND**, not only are the two equivalent, but if only the latter three calls to **assert-ind** were made, CLASSIC would be able to answer affirmatively a query about **Rocky**'s being a RICH-KID.

Databases will want to record some information involving numbers, strings, and other 'primitive' data values. Without special provisions, these would have to be encoded as individuals, creating them the first time we see them, and keeping a dictionary of their names—clearly an onerous task. Furthermore, we would like to use CLASSIC eventually for computer-aided software engineering, to maintain software objects in the database. For this reason, we opted to build into the CLASSIC language a fundamental distinction: every individual known to the database needs to be either a **host** individual—a valid value from the space of values of the host implementation language (LISP or C in our case)—or a **regular** (**CLASSIC**) individual. Host individuals cannot have roles, but are otherwise first class citizens—they can be grouped by enumerated concepts, for example.

All of the updates discussed so far work in the obvious fashion when they are monotonic—when they do not contradict earlier assertions. We have also formulated and are now implementing a facility for making "destructive updates" (i.e., ones that contradict earlier assertions) and will report on this at a future date.

3.3 CLASSIC as an active database

The fact that concepts, including those named and used in the schema, can be *defined* in terms of the properties that must be satisfied by their instances clearly allows a CLASSIC database to 'recognize' new instances of a concept. So, for example, if **STUDENT** were defined to be

```
(AND PERSON
     (AT-LEAST 1 enrolled-at)),
```

then the moment we learn that **Rocky** (who was previously asserted to be a **PERSON**) is enrolled at some school we implicitly recognize **Rocky** as a **STUDENT**—it is not necessary to explicitly assert this fact. In other words, without seeing any explicit mention about the individual **Rocky**'s relationship to the concept **STUDENT**, CLASSIC would still include **Rocky** in the answer to a query about the instances of **STUDENT**.

On the other hand, one can, if one wants to, assert that a complex concept description holds of an individual; this allows the DB to deduce a number of new relationships. For example, co-reference constraints in a concept can lead to roles being filled by new values:

```
assert-ind[Rocky,
           (SAME-AS (likes) (thing-driven))]
```

would lead to **likes** being filled by **Volvo-17**, if it were already known that **Rocky** drives **Volvo-17**. Further, **AT-MOST** restrictions on roles can allow the DB to deduce that a role is closed:

```
assert-ind[Rocky, (AT-MOST 1 thing-driven)]
```

results in **thing-driven** being closed as soon as we learn that **Rocky** drives **Volvo-17**.

In addition to these kinds of inferences, we have added a limited form of forward chaining—the ability to specify *rules* of the form 'if an individual is a < *concept*$_1$ > then it is also a < *concept*$_2$ >' :

```
assert-rule[STUDENT, (ALL eat JUNK-FOOD)]
```

adds to the database the information that any instance of **STUDENT** is also an instance of the class of things that eat only junk food. Note that this is very different from making **(ALL eat JUNK-FOOD)** part of the *definition* of **STUDENT**, because in that case, someone would not be recognized as a **STUDENT** until it was known that she also ate junk food, whereas the current rule allows the DB to *deduce* that she eats junk food as soon as we know she is enrolled at a school (and hence is a **STUDENT**).

Note also that we do not consider rules to be equivalent to logical implications; they are more appropriately viewed as triggers activated only when a new individual is found of which the antecedent concept description holds.

3.4 Integrity checking in CLASSIC

When asserting that an individual is describable by some concept, it is of course necessary to verify that the previously asserted facts about the individual do not contradict this description. For example, we cannot have an individual belong to a concept that contains the description **(AT-MOST 0 thing-driven)** and at the same time have asserted that its **thing-driven** role is filled by some object. Therefore integrity checking in a CLASSIC database involves verifying the consistency of concept descriptions and role fillers.

Experience indicates that there are many situations where primitive subclasses being defined from a common parent class are known to be disjoint. For example, **MALE** and **FEMALE**, **TALL** and **SHORT**, etc., are all mutually exclusive primitive subsets of **PERSON**. CLASSIC allows this to be expressed as an integrity constraint through the *disjoint primitive* variant of the **PRIMITIVE** constructor. Its use is illustrated by the definition of the **MALE** and **FEMALE** subclasses of **PERSON**:

```
(DISJOINT-PRIMITIVE PERSON gender male)
(DISJOINT-PRIMITIVE PERSON gender female)
```

where **gender**, **male**, and **female** are simply identifiers, whose importance lies only in their distinctness. (**Gender** names the grouping of disjoint concepts, and **male** and **female** distinguish the elements within this grouping.) More than one disjoint grouping is allowed below a single concept, e.g., **PERSON** could be grouped by both gender and age.

3.5 Querying the DB

A CLASSIC database can be viewed as maintaining three types of relationships: those between concepts, those between individuals, and those between an individual and the concepts that describe it.

3.5.1 Queries about concepts

Because concepts are usually built up through the conjunction of other concepts, it is useful to be able to look at various aspects of a concept contributed through the different constructors. The **concept-aspect** operator allows one to look at these facets, by taking as arguments a concept, a constructor, and possibly a role name. For example,

- **concept-aspect[c, ONE-OF]**
  inquires about any enumeration that is part of the definition of the concept **c**;
- **concept-aspect[c, ALL, thing-driven]**
  returns the type constraint on the **thing-driven** role fillers that has been imposed on instances of **c** by its definition;
- **concept-aspect[c, AT-LEAST, thing-driven]**
  returns the lower bound constraint on the **thing-driven** role that has been imposed on instances of **c**.

For convenience, by dropping the role argument we get the list of roles for which there is a restriction present:

concept-aspect[c, ALL]

returns all roles that have been restricted by **ALL** constraints for **c**.

It is also possible to find out certain relationships between concepts. The most fundamental one is *subsumption*: if C1 and C2 are concepts, then **concept-subsumes**[C1, C2] is true if and only if in every state any individual satisfying C2 is *necessarily* (i.e., by definition) also an instance of C1. Two concepts are equivalent if and only if they subsume each other. Many other operators can be defined on the basis of **concept-subsumes**. For example, the subsumption relationship induces an acyclic directed graph over the space of named concepts—the (in)famous 'IS-A hierarchy'.[5] Among other things, it is then often useful to know which *named concepts* (those in the schema) are the most specific subsumers or subsumees of some concept—the 'immediate parents' or 'immediate children' of the concept in the IS-A hierarchy.

3.5.2 Queries about role fillers

The facts asserted about an individual's relationship to other individuals through roles constitute what would be an ordinary database: just consider each role as a binary relation, and every primitive concept as a unary relation, and one has an ordinary relational database (modulo the closed world assumption). Similarly, by considering roles as set-valued functions, and primitive concepts as classes, one can easily view these facts as a functional database [27]. Presumably, much of what has been learned about querying relational or functional databases could be applied to this problem, although it would be important to take into account our open world assumption. We have not spent much effort in devising an elaborate query language for this space of facts—at the moment it is possible to ask for all the fillers or restrictions of a role for an individual, and whether it is closed or not, by using the **ind-aspect** operator, which behaves similarly to **concept-aspect** but in addition recognizes the invocations **ind-aspect**[<ind>, **FILLS**, <role>] and **ind-aspect**[<ind>, **CLOSE**, <role>]. We plan to develop a more powerful and integrated query language, possibly based on the notation used in [5].

3.5.3 Queries about class membership

An arbitrary concept expression can be viewed as a query requesting information about all the individuals in the DB that satisfy it (i.e., individuals that are members of the class defined by the concept). Thus, **PERSON** asks for all individuals known to be instances of the primitive class **PERSON**. The query **(AT-LEAST 2 thing-driven)** would return all individuals that have at least two specific role fillers for **thing-driven**, *or* whose concept descriptor *entails* that they have at least 2 fillers for this role (e.g., a concept descriptor which contains **(AT-LEAST 3 thing-driven)**). Therefore concepts used as queries take into account the structural descriptions asserted of individuals.

We generalize this notion of query and answer in several ways. First, concepts allow us to qualify an object by describing its role fillers. It seems reasonable to ask for qualifications regarding the role an object itself fills. A simple way of accomplishing this, without introducing inverse roles, is to allow the query concept to distinguish a specific subexpression, whose individual instances are desired. Using **?:** as this marker, the query concept

```
(AND STUDENT
     (ALL thing-driven
          ?:(ALL maker (one-of Ferrari))))
```

can then be interpreted as asking for the objects that are driven by students and have **maker Ferrari**. The earlier query for all persons would simply be **?:PERSON**.

Finally, once we assume that the database does not have complete knowledge of the world, it becomes reasonable to ask for information that *necessarily holds of all possible individuals* that satisfy the query—not just those currently known, but those that *might* be added to the DB in time. This of course becomes interesting once we have rules, because these supply additional information not contained in the original query. So

```
(AND STUDENT (ALL eat ?:THING))
```

can be interpreted as asking for a concept describing all objects that are eaten by students, independent of the known examples—the description of this set, in light of the forward-chaining rules in effect at that time, might include **JUNK-FOOD**; also included would be any other facts that the DB might infer about the query concept (and its role fillers up to the **?:** marker) through rules and reclassification.

One way to accommodate all of the above versions of querying is to provide the operators **ask-necessary-set**, and **ask-description**; these take as argument a query concept, which is just a concept with a single **?:** embedded as a distinguishing marker, placed in front of some subexpression.

4 A brief example

The following examples illustrate the integration of the various features of CLASSIC presented earlier. Consider a database about crimes and criminals that might be maintained by some law-enforcement agency. This is a typical situation where one starts out with an incomplete view of the actual events, and incrementally fleshes out the details of the crime, and especially the perpetrators and their identities.

A central primitive concept would be **CRIME**; every crime would need to have at least one perpetrator, who is a person, some victim(s) (these need not be persons!), and a site. A person would have a domicile. To model this, we would define **CRIME** to be a primitive concept which, among other things, has **victims**, **perpetrators**, and a **site**:

```
define-role[perpetrator]
define-role[victim]
...
define-concept[CRIME,
      (PRIMITIVE
          (AND (AT-LEAST 1 perpetrator)
```

[5] It is crucial to realize that for non-primitive concepts the IS-A hierarchy is *induced* by the definitions, and is not an independent structure under control of the user.

```
                  (ALL perpetrator PERSON)
                  (AT-LEAST 1 victim)
                  (AT-LEAST 1 site)
                  (AT-MOST 1 site))
              crime)].
```

When a new crime occurs, an object for it is added to the database:

```
create-ind[crime23, CRIME],
```

and as evidence accumulates, more information about it is added. This is where the open-world assumption and ability to use structural descriptions are obviously useful. For example, if a witness saw a group of criminals leaving, we would invoke

```
assert-ind[crime23,
           (AT-LEAST 2 perpetrator)].
```

And if these people were overheard to be speaking Ruritanian, then in addition we could invoke

```
assert-ind[crime23,
           (ALL perpetrator
                (ALL heard-speaking
                     (ONE-OF Ruritanian)))]
```

after first creating the role **heard-speaking**, if necessary. (Note the usefulness of the ability to create new attributes—and hence extend the schema—on the fly: it seems hard to anticipate all possible kinds of clues to crimes.) As the identities of the criminals are discovered, they are entered as filling the **perpetrator** role of **crime23** using the **FILLS** constructor.

There are a variety of kinds of crime, for which concepts could be defined. For example, domestic crime might be defined as a crime perpetrated at the domicile of the (single) perpetrator:

```
define-concept[DOMESTIC-CRIME,
       (AND CRIME
            (AT-MOST 1 perpetrator)
            (SAME-AS (site) (perpetrator
                                domicile)))].
```

Note that it is inferrable by CLASSIC that a DOMESTIC-CRIME has *exactly* one perpetrator because it has CRIME included in its definition, and every CRIME has at least one perpetrator.

One could also maintain in the database information about the typical suspect for crimes, through role **typical-suspect**; in this case, rules of a heuristic nature could be added to the knowledge base, such as "domestic criminals are typically adults, and have no jobs", which might be encoded as the rule

```
assert-rule[DOMESTIC-CRIME,
            (ALL typical-suspect
                 (AND ADULT
                      (AT-MOST 0 jobs)))].
```

When used as a query with **ask-necessary-set**, the concept ?:DOMESTIC-CRIME would return all instances of such crimes in the database (including ones where the identity of the perpetrator is not yet known exactly: "did the wife or husband do it?"); on the other hand (AND DOMESTIC-CRIME (ALL **perpetrator** ?:THING)) would be used to ask for perpetrators of domestic crimes.

The **ask-description** operator could be applied to some crime **crime15**, as in

```
ask-description
       [(AND (ONEOF crime15)
             (ALL typical-suspect ?:PERSON))],
```

to see if **crime15** was classified as a kind of crime for which additional descriptive information about its suspect can be inferred, and to show this information.

Systems like CLASSIC have been used in real-world applications involving retrieval of information from complex structured domain models. KANDOR, the immediate predecessor of CLASSIC, has been used to implement a prototype tool for representing and querying a knowledge base of several hundred concepts (and several thousand individuals) about a large software system and its structure. [12] The knowledge base for this system has already been upgraded to use CLASSIC, and new tools are being designed to exploit CLASSIC's deductive power and completeness.

5 Comments on implementation and tractability

The current prototype implementation of CLASSIC (which is written in CommonLisp, and is being rewritten in C) opts for a great deal of preprocessing in order to facilitate query answering. Among other things, all concepts in the schema are reduced to a normal form, and then are compared to each other to establish the subsumption hierarchy. The subsumption relationship is established in time proportional to the sizes of the two concepts. Individuals are similarly normalized and are classified whenever new information about them is asserted, so that each individual is associated with the lowest concept(s) in the schema whose description(s) it satisfies. This might cause other individuals to be reclassified, but this process is guaranteed to end because it is bounded by the number of classes and individuals in the database: every individual can move into a class at most once (since there is no 'removal').

Query answering follows the technique introduced in [25]: first, the query concept is itself 'classified' with respect to the concepts in the schema;[6] then the instances of the parent concepts are tested individually to see if they satisfy the query concept. The advantage of this technique is that all instances of schema concepts that are subsumed by the query are known to satisfy the query and are therefore not explicitly tested. Assuming that the schema can fit in main memory, this approach will reduce disk access traffic in the case of large databases.[7]

Each rule is associated with a specific schema concept and the rule application is triggered whenever an individual becomes an instance of that class. Rules continue propagating until a fixed point is reached.

The complexity of the algorithms for performing the above operations is very sensitive to the choice of concept constructors in the language. These tractability issues were first recognized in [8], and over the last several years we have explored extensively the computational behavior of systems like CLASSIC. [19, 24] We have made a concerted effort to keep the CLASSIC language small and eliminate 'expensive' features. This explains the absence of such obvious concept constructors as **OR** and **NOT**, but has also led to more subtle restrictions such as the requirement that co-reference constraints be used only with roles that are single-valued. We note that our current algorithm for subsumption has low-order polynomial complexity and we believe that it is complete, though a formal proof is not yet available.

A major research issue for the future is what algorithms and data structures are best suited to implementing CLASSIC

[6] Classification is the operation by which all known subsuming and subsumed concepts are found.

[7] A more detailed analysis of some ideas for implementing such classification systems appears in [20].

once we assume that there are very many concepts and individuals around, possibly requiring secondary storage. Some ideas for efficiently maintaining information about the subsumption hierarchy itself appear in [2].

6 Conclusions and related work

The work presented here lies at the confluence of several different approaches to the problem of managing information.

To begin with, CLASSIC belongs to the family of knowledge representation languages that are descendants of KL-ONE [9], including languages and systems such as KANDOR [23], LOOM [20] and BACK [29]. These systems all offer sublanguages for defining terms such that the subsumption relationship *holds by definition.* The CLASSIC concept language resembles them to a considerable extent, but differs from them in one or more of the following features: enumeration using **ONE-OF**, integration of host objects, test concepts, the presence of rules (and complete propagation to a fixed point), the general treatment of incomplete information through ideas such as the open-world assumption, concept constructors for individuals, and multiple ways of querying the DB, including descriptions of answers.

There is also a considerable literature on the description of views and meta-data (e.g., [21]) which has a bearing on our work in the sense that new concepts (rather than instances) are *defined* from others. However, to our knowledge such systems do not reason with the newly defined concepts in terms of their relationship to each other, nor do they apply rules, etc., to them.

6.1 Object-based models and hierarchies

Semantic networks and semantic data models [16]—their descendants in databases—also resemble CLASSIC, especially in their object-oriented world-view and their exploitation of subclass hierarchies. The IFO model [1], in particular, has considered the 'structure' of the data. But this body of work is more distantly related since classes are almost always treated as primitive identifiers, as opposed to definitions that can recognize instances, and that can be composed and related to each other.[8] So, for example, although many semantic data models have constructs for specifying upper and lower bounds on the number of role fillers, these act only as integrity constraints on the data stored, rather than allowing one to recognize instances of that class.

Object-oriented databases [13] are in many ways similar to semantic data models, or languages based on them, but may in addition concern themselves with the problems of methods/message passing, which are foreign here. The earlier remarks on semantic data models apply equally well to these.

The work of Aït-Kaci [3] on term-structures with coreference was a source of inspiration for aspects of the CLASSIC language relating to equality. But Aït-Kaci's term structures assumed that all roles were single-valued, and generalized this using the notion of equivalence classes. Also, rules and deduction in general are not part of the work on structured terms—instead Aït-Kaci has chosen to merge Prolog with structured terms used as types [4].

6.2 Non-traditional queries and answers

Probably most closely related to CLASSIC is the work on the new data model CANDIDE. [7] It essentially has many of the features of the terminological logics noted above (being based on KANDOR), but is phrased in a notation more familiar to database researchers, and is extended for expressive ease. We support and agree with many of the arguments presented in [7], including the specification of classes with definitional meaning, and the merging of the DML and DDL through the use of concepts as queries. We also share a concern for computational tractability. Unfortunately, it is now known that KANDOR (and thus CANDIDE) has a trap where complexity gets out of hand [22]. CLASSIC has tried to avoid this trap, and has also added new features, such as using concepts as partial descriptions of individuals both in updates and queries, and adding rules and co-reference constraints.

On the other hand, the issue of non-enumerative answers—answers that consist of descriptions of the elements in the answer set—has received some attention in the literature on expert database systems (e.g., [11, 18, 28, 15]). Several general approaches can be discerned:

1. Using the rules in a backward-chaining manner to derive from a query a (possibly empty) list of actual individuals plus a condition on the remaining answers expressed using some pre-specified *subset* of the predicates. For example, the answer to 'Who can teach calculus?' might include 'Anyone who can teach the prerequisites of measure theory'. This is useful, among other things, as a partial answer if one runs out of time in query processing.
2. Using the current extensions of certain database predicates to characterize the answer set. For example, the answer to 'Who earns over $50,000?' might be: 'Persons who are tenured and have been chairpersons.' This is useful if the answer is too long, and if the current set of individuals is representative of the class.

The first of these techniques considers *sufficient* conditions for some object—known or not yet known—to be an answer, while the second one characterizes the *current* extensional answer set. Our idea is to provide *necessary* conditions that describe and need to hold of the objects which are in the answer set—conditions that are a consequence of satisfying the query concept.

Also related is the work of Goebel [15] on the manipulation of definite descriptions of objects and sets (e.g., "the x such that ..."), since these are also intensional descriptions. These can be manipulated as terms in an extension of Prolog, thus providing considerable generality and power to the language, partly because it is connected to Prolog as a data manipulation language, and partly because the reasoning with these definite terms is carried out as full *general-purpose theorem proving*, using predicates like **DEMONSTRATE**. This is very different from our much more limited approach in CLASSIC.

To summarize, we believe that CLASSIC contributes novel ideas to the solution of the problem of knowledge management on several fronts:

1. Allowing the database to be viewed as a potentially incomplete model of the world:
   - individuals can be described not only in terms of their relationship to other individuals, but also in terms of their 'conceptual structure' (e.g., 'has 4 brothers', 'has brothers who are all doctors');
   - features such as the absence of the closed world assumption support an incremental model of information acquisition.
2. Allowing the database to actively discover a limited number of new relationships between individuals, not explicitly asserted by users:

[8] The significant exception is the CANDIDE data model, discussed below.

- concepts are classified with respect to each other, and individuals are classified under concepts specified in the schema;
- concept constructors (such as numeric bounds, co-reference constraints, and enumerations) can add information about role fillers of individuals;
- simple forward chaining rules provide new descriptors for individuals.

3. Providing a *simple and uniform interface* to the database:
   - through the use of multiple operators, a *single language* is used to specify the schema (including integrity constraints), the information added to the database, and the queries to it;
   - the schema and data can be manipulated uniformly and with 'closure': schema objects (concepts) can be created, queried and obtained as answers at any time;
   - as a result, the description of the entire interface is brief—it appears here as a short appendix.
4. Providing a variety of notions of 'queries and answers':
   - three kinds of relationships can be queried: a standard one concerning the inter-relationship of individuals, one concerning the relationship of concepts, and, most interestingly, one relating concepts and individuals; it is this latter facility that takes into account incomplete information;
   - because of the open world assumption, different kinds of answers to queries can be considered: sets of individuals that are known to satisfy the query, sets of individuals that might satisfy the query, and a most-specific description of the necessary properties of the objects, known or unknown, that might satisfy the query.

We consider as a significant contribution the establishment of a framework that treats both incomplete information and active deduction in a manner different from approaches based on first order logic. Although the contents of a CLASSIC database can be expressed in ordinary logic, the point we are making is that languages such as CLASSIC impose syntactic restrictions on the form of the formulas permitted that are different from those that arise normally in FOL. So, although the preceding list of features might seem very ambitious, hence prohibitively expensive, the power of CLASSIC resides largely in the particular language of structured concepts, and secondarily on the form of the rules; it is thus possible to avoid computational intractability by carefully limiting the specific features included in this language. Whatever the shortcomings of our particular language, we now have a framework in which to search for inferential data models that balance expressiveness with tractability.

Appendix A: The grammar of the CLASSIC language

```
<concept-expr> ::=
   THING | CLASSIC-THING |              [these three are
   HOST-THING |                         built-in primitives]
   <concept-name> |
   (AND <concept-expr>+)[9] |
   (ALL <role-expr> <concept-expr>) |
   (AT-LEAST <positive-integer> <role-expr>) |
   (AT-MOST <non-negative-integer> <role-expr>) |
   (SAME-AS (<role-expr>+) (<role-expr>+)) |
   (TEST <fn> <realm>) |
   (ONE-OF <individual-name>+ ) |
   (PRIMITIVE <concept-expr> <index>) |
   (DISJOINT-PRIMITIVE
        <concept-expr> <partition-index> <index>)
<individual-expr> ::=
   <concept-expr> |
   (FILLS <role-expr> <individual-name>) |
   (CLOSE <role-expr>) |
   (AND <individual-expr>+ )
<realm>             ::= HOST | CLASSIC
<concept-name>      ::= <symbol>
<individual-name>   ::= <symbol> | <host-lang-expr>
<role-expr>         ::= <symbol>
<index>             ::= <number> | <symbol>
<partition-index>   ::= <number> | <symbol>
<fn> ::= a unary function with boolean return type
         that can be evaluated in the host language.
```

Appendix B: Operators on a database

```
UPDATE
  define-role[<id>]
  define-concept[<id>, <concept-expr>]
  assert-rule [<concept>, <concept-expr>]
  create-ind[<id>] | create-ind[ ]
  assert-ind[<ind>, <individual-expr>]
QUERY
  concept-aspect[<concept-expr>, <constructor>] |
  concept-aspect[<concept-expr>,
                          <constructor>, <role>]
  ind-aspect[<individual-expr>, <iconstructor>] |
  ind-aspect[<individual-expr>,
                      <iconstructor>, <role-expr>]
  concept-subsumes[<concept-expr>, <concept-expr>]
  ask-necessary-set[<query-concept>]
  ask-description[<query-concept>]
where
  <constructor> ::= AND | ALL | AT-LEAST |
     AT-MOST | SAME-AS | TEST | ONE-OF|
     PRIMITIVE | DISJOINT-PRIMITIVE
  <iconstructor>    ::= <constructor> | FILLS | CLOSE
  <query-concept> ::= ?:<concept-expr>
```

References

[1] Abiteboul, S., and Hull, R. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, December 1987.

[2] Agrawal, R., Borgida, A., and Jagadish, H. V. A transitive closure compression technique. To appear in Proc. ACM SIGMOD'89 Conference.

[3] Aït-Kaci, H. *A lattice-theoretic approach to computation based on a calculus of partially-ordered type structures.* PhD thesis, University of Pennsylvania, 1984.

[9] + means one or more values separated by blanks

[4] Aït-Kaci, H., and Nasr, R. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:187–215, 1986.

[5] Anderson, T. L., Ecklund, E., and Maier, D. PROTEUS: Objectifying the DBMS user interface. In Dittrich, K., and Dayal, U., editors, *Proc. 1986 International Workshop on Object-Oriented Database Systems*, Asilomar, CA, 1986. IEEE Computer Society Press.

[6] Bancilhon, F., and Ramakrishnan, R. An amateur's introduction to recursive query processing. In *Proc. ACM SIGMOD'86 Conference*, pages 16–52, May 1986.

[7] Beck, H. W., Gala, S. K., and Navathe, S. B. Classification as a query processing technique in the CANDIDE data model. In *Proc. Fifth International Conference on Data Engineering*, pages 572–581, Los Angeles, CA, 1989.

[8] Brachman, R. J., and Levesque, H. J. The tractability of subsumption in frame-based description languages. In *Proc. AAAI-84*, pages 34–37, 1984.

[9] Brachman, R. J., and Schmolze, J. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April–June 1985.

[10] Brachman, R. J., Borgida, A., McGuinness, D. L., and Resnick, L. A. The CLASSIC knowledge representation system, or, KL-ONE: The next generation. In preprints of *Workshop on Formal Aspects of Semantic Networks*, Santa Catalina Island, California, February 1989.

[11] Cholvy, L., and Demolombe, R. Querying a rule base. In *Proc. First International Conference on Expert Database Systems*, pages 365–371, 1986.

[12] Devanbu, P. T., Selfridge, P. G., Ballard, B. W., and Brachman, R. J. Steps toward a knowledge-based software information system. Submitted to Eleventh International Joint Conference on Artificial Intelligence, 1989.

[13] Dittrich, K., and Dayal, U., editors. *Proc. International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986. IEEE Computer Society Press.

[14] Gallaire, H., Minker, J., and Nicolas, J-M. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16(2):153–186, June 1984.

[15] Goebel, R. G. The design and implementation of DLOG, a Prolog-based knowledge representation system. *New Generation Computing*, 3:385–401, 1985.

[16] Hull, R., and King, R. Semantic database modelling: survey, applications, and research issues. *ACM Computing Surveys*, 19(3):210–260, September 1987.

[17] Imielinski, T. Incomplete deductive databases. Rutgers University. Unpublished manuscript, 1987.

[18] Imielinski, T. Intelligent query answering in rule based systems. *Journal of Logic Programming*, 4(3), September 1987.

[19] Levesque, H. J., and Brachman, R. J. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3(2):78–93, May 1987.

[20] MacGregor, R. F. A deductive pattern-matcher. In *Proc. AAAI-88*, pages 403–408, St. Paul, MN, August 1988.

[21] Mark, L. Defining views in the binary relational model. *Information Systems*, 12(3):281–294, 1987.

[22] Nebel, B. Computational complexity of terminological reasoning in BACK. *Artificial Intelligence*, 34(3):371–383, April 1988.

[23] Patel-Schneider, P. F. Small can be beautiful in knowledge representation. In *Proc. IEEE Workshop on Principles of Knowledge-Based Systems*, pages 11–16, Denver, December 1984. A revised and extended version is available as AI Technical Report Number 37, Schlumberger Palo Alto Research, October 1984.

[24] Patel-Schneider, P. F. *Decidable, Logic-Based Knowledge Representation.* PhD thesis, Department of Computer Science, University of Toronto, February 1987. A slightly revised version available as Technical Report Number 56, Schlumberger Palo Alto Research, May 1987 and as Technical Report 201/87, Department of Computer Science, University of Toronto.

[25] Patel-Schneider, P. F., Brachman, R. J., and Levesque, H. J. ARGON: Knowledge representation meets information retrieval. In *Proc. First Conference on Artificial Intelligence Applications*, pages 280–286, 1984.

[26] Reiter, R. On closed world databases. In Gallaire, H., and Minker, J., editors, *Logic and Databases*, pages 55–76. Plenum Press, 1978.

[27] Shipman, D. W. The functional data model and the data language DAPLEX. *ACM Trans. on Database Systems*, 6(1):140–173, March 1981.

[28] Shum, C. D., and Muntz, R. Implicit representation of extensional answers. In *Proc. Second International Conference on Expert Database Systems*, pages 257–273, 1988.

[29] von Luck, K., Nebel, B., Peltason, C., and Schmiedel, A. The anatomy of the BACK system. KIT-Report 41, Technische Universitat Berlin, January 1987.