ELSEVIER Data & Knowledge Engineering 14 (1994) 203–249

DATA & KNOWLEDGE ENGINEERING

CoLan: A functional constraint language and its implementation

N. Bassiliades*, P.M.D. Gray

Dept. of Computing Science, University of Aberdeen, King's College, Old Aberdeen, AB9 2UE, UK

Received 19 November 1993; revised 7 March 1994; accepted 5 October 1994

Abstract

This paper is about the definition of CoLan, a high-level declarative Constraint Description Language, for use with an Object-Oriented Database (OODB). CoLan has features of both first-order logic and functional programming and is based on Daplex. CoLan expressions are translated into Prolog code that implements the operational semantics of the constraint. Pieces of generated code are cached inside the class descriptor of the 'host' class attached to appropriate slots. The pieces of code are retrieved along an inheritance path when an update on the database is attempted. If the update violates any of the retrieved constraints then it is rejected with an informative message. Thus constraints are expressed declaratively and they can even be retracted individually. However, they are implemented efficiently as code-generated methods, triggered selectively by an update. The implementation is described for the ADAM OODB, which uses meta-classes of the CoLan system to generate class descriptions.

Keywords: Semantic integrity constraints; Object-oriented databases; Functional data model; Constraint compilation; Incremental constraint checking; Numerical quantifiers

1. Introduction

The goal of storing integrity constraints as part of the database system, instead of having them embedded in applications, has been pursued ever since the late 70's. In particular, Nijssen [21] enunciated the principle that 100% of such constraints should be in the database. Older programming languages made this hard to achieve, because it was hard to add extra procedures to running application code in C or PL/1. This paper describes a semantic constraint language CoLan for an object-oriented database (OODB), implemented using Prolog procedures as stored methods on entity classes. Thus we are not relying on the

* Corresponding author. Current address: Dept. of Informatics, Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece. Email: bassiliades@olymp.ccf.auth.gr

SSDI: 0169-023X(94)00028-X

deductive features of Prolog as in deductive databases. We do use unification and pattern matching for query transformation [13, 17], but the main advantage of object-oriented construction using a garbage-collected language such as Prolog or SmallTalk is that it is comparatively easy to add new procedures, to store them in data structures, and to bind them to existing code. This surely is the right way ahead for advanced database systems capturing modern semantic data models.

CoLan captures a wide range of semantic constraints that are not expressible declaratively by commonly used Data Description Languages, because they are not based on a semantic data model. These are predicate constraints which restrict the allowable states of the database. For example, one can specify that a patient in a hospital only takes drugs where they have no allergy to any of the drug components:

```
forall p in patient
  forall c in components of drugstaken of p
    c not in allergies of p
```

One can also specify existence constraints with numerical quantifiers, for example to say that each patient has at most two doctors:

```
forall p in patient
  exist at most 2 d in doctor of p
```

However, the constraints are not dynamic, in the sense that they restrict allowable states but not particular paths used in transitions between them; also they do not currently check post conditions relating to values in old and new states.

CoLan is significant because it allows one to express constraints declaratively, and to add or retract constraints incrementally, whilst taking advantage of an efficient implementation technique which uses methods triggered by an update. Thus it has the speed advantage of code embedded in a method, but since it is generated from a declarative form stored in a database, then it can always be altered and regenerated. Furthermore, its formal specification is readable and can be referenced by way of explanation or used by a query optimiser. Thus it overcomes many of the objections made in [27], which argues strongly against embedding constraints in unintelligible method code. Instead, CoLan uses constraint methods which are effectively cached with each class and inherited by subclasses.

CoLan uses a functional style, instead of the inference rule style used by the ALICE language [30]. This is because we are looking for readability. It is strongly influenced by the syntax of Daplex [26], which we use in one of the databases for which CoLan is designed. However, it has a similar underlying view of data to many semantic data models, including that used by ALICE. Thus, property definitions are viewed as functions defined over entity classes. The functions may be single or multi-valued (i.e. return a set or sequence). They may range over basic atomic types or else over entity types, and thus be used to represent relationships between entities. The functions may represent a stored relationship, such as `suppliers(drug)`, but they may instead be implemented by calculation using a stored procedure (or method). The entity classes can form part of a subtype hierarchy, in which case all properties and methods on the superclass are inherited by each subclass.

This similarity in basis makes CoLan portable to a range of object-oriented data models. In this paper we describe how it has been implemented as an extension to the ADAM data language [13, 22]. It provides ADAM with a declarative front end and thus saves the user from writing many separate complex pieces of method code. Instead, it code-generates pieces of Prolog which are activated by messages sent to class descriptors. It also uses the powerful meta-class descriptors of ADAM to ensure that constraints on superclasses cannot be overridden by more specialised constraints – these can only be added on in conjunction.

The constraint language allows various schemes for its enforcement. In the case of ADAM we have taken the path of code-generating methods for each class on each slot name (i.e. attribute or relationship) that appears in a constraint. Each such method starts by calling a system predicate which checks inherited constraints. Thus the bodies of the various constraint methods are actually executed starting at the top of the supertype hierarchy. If something different is wanted, then as argued by [25] one could instead represent the constraints by active rules and call out to an active rule interpreter. In the case of P/FDM, a large object-oriented system based on the functional data model [13], performance is critical, so we intend to use the generated method approach as described in this paper, but with a more complex transaction model [10].

Besides using object-oriented construction, CoLan relies on the structural constraints of a semantic data model schema; it is not intended as a language for stating these constraints. In particular, it does not make checks on single-valuedness or totality of functions, and on disjointness or covering properties of subtypes, since these are done more efficiently within kernel system code. What CoLan checks are semantic constraints. These are expressed as Boolean-valued expressions involving sets and functions taken from the schema, whose values must be maintained true for any change in the number or state of objects belonging to a given entity class (or classes). Variables in the set expressions always range over a finite known universe, either a set of object identifiers of stored objects, or a subrange of integers. Thus we do not have the problems of safety [32] that happen with relational calculus formulae.

The remainder of this paper is structured as follows; In Section 2 we overview related approaches to constraint description. Section 3 presents CoLan language. In Section 4 we give a background to ADAM, the OODBMS that CoLan was first implemented. The next sections describe the implementation of CoLan; Section 5 describes the structure in detail and Section 6 gives a formal model of the language which is used for the generation of code. Section 7 addresses some of the deficiencies of the current CoLan implementation. Finally, Section 8 concludes this paper with a summary of our work and future directions for CoLan.

2. Related work

Many people [6, 19, 20] have suggested that constraints should be expressed in pure logic and held in deductive databases, so that the constraints can be proved correct by some kind of mechanised proof technique. However, semantic constraints on explicit data are relatively straightforward and deterministic to check. They apply to specific instances and have to be checked individually. Thus there is no particular advantage to a logic representation for CoLan, and indeed we claim that the functional form is easier to read. However, it would be useful to have a logic-based checker to show that a new constraint being inserted was not inconsistent with existing constraints, nor was it subsumed by any of them. This would improve performance. It is really part of constraint design and analysis.

Kulkarni and Atkinson [18] describe a syntax for constraint definitions in their implementation of Daplex (EFDM). Constraints are expressed on property values thus:

```
constraint c1 on cname(person), sname(person)→unique;
constraint c2 on sex(person)→total;
constraint c3 on student, staff→disjoint;
constraint c4 on grade(student,course)→
  some c in courses(student) has c=course
```

Of these, c2 and c3 are structural constraints that CoLan is not designed to handle, as mentioned earlier, c1 is a structural constraint but expressible in CoLan, and c4 is a predicate constraint of the kind which CoLan is designed to express. One weakness of EFDM constraints is that they are expressed in terms of restricting the values of specific properties on specific entity types. Instead, CoLan expresses the constraint on the entity class rather than the property, and although stored with this class it is cross-referenced from all the update-methods for properties mentioned in the constraint. Thus one constraint stands for several constraints in EFDM. Thus we should express c4 in CoLan by introducing an enrollment entity with properties student, course and grade:

```
forall e in enrollment
  exists c in courses of student of e
    such that c=course of e
```

Constraint c1 can be expressed as:

```
forall p in person
  not exists p2 in person
    such that cname of p2=cname of p
      and sname of p2=sname of p
```

Note that CoLan implicitly requires that a second variable over the same entity class does not take the same value as the first (i.e. p<>p2). CoLan generalises the EFDM syntax in that it quantifies over entity identifiers and not over property values, also it generalises the notion of 'fixed' value by using the comparator exactly. It also implements numerical quantifiers, such as at most 3, which appear in EFDM syntax but not in any published examples of EFDM constraints.

Urban [29, 30] discusses many types of constraint. As noted in [13] most of these are enforced by FDM as structural constraints, apart from the unique constraint discussed above and a covering constraint for subtypes. The ALICE language is used by [30, 31] to write constraints in a predicate logic style. However, we believe that constraints are easier to read in a function style. The functional form also makes it natural to call out to functions which compute derived values, for example using methods declared on object classes. There are no examples of constraints incorporating derived values in [31]. Thus we feel that the functional form for constraints fits better with an object-oriented approach.

The original use of ALICE was for 'constraint analysis', in conjunction with tools which allowed a user to foresee interactions between constraints and to specify actions to recover from violations. Thus there was no system to enforce the constraints, but in [31] a system is described for constraint enforcement through active rules. This system generates methods ('state-altering database operations') which trigger active rules generated from ALICE constraints. This is an impressive system. Its aims are similar to ours in that it code-generate rules from declarative specifications. The main difference is that it uses an active-rule interpreter, with a forward chaining mechanism instead of a recursive Prolog goal-driven mechanism. Their system also attempts to use rules to recover from integrity violation by triggering other updates. This can run into cycles or 'anomalous rule behaviour'. By contrast, we are concerned only with rejecting invalid transitions, which is an easier problem to solve.

A recent paper on ADAM by Paton, Diaz and Barja [25] argues strongly for using a combination of metaclasses and active rules in ADAM to represent object semantics. They are sceptical of the value of enforcing constraints through methods. In particular, they are worried about having specialisations of methods accidentally overriding important constraints. They are also worried about unforeseen interactions between constraints inherited from methods defined independently on various superclasses. They propose to solve this by triggering active rules from the methods (which has also been done by [5] for a relational database), and passing the problems to an active rule interpreter for it to resolve the interactions. We believe we can overcome their objection by code-generating all constraints in a way that cannot be overridden. Thus we insist that constraints are defined in a high-level language which is easier to check at compile time and in a single consistent regime, which makes it easier to deal with interactions.

Another system [16] that has recently proposed a design for generating constraint checks as part of methods is Ode. They propose a language CIAO++ with quantifiers similar to ours, but copying the C++ syntax for record selectors and boolean expressions. Their syntax for quantifiers uses: `foreach X in C` and `thereis X in C`, but does not include numeric quantifiers as in CoLan such as: `exist at least 5 X in C`.

They have the same aim of code generating triggers from a declarative specification of the constraints, but using C++ for performance gains. The really significant difference is that they have no model for constraint inheritance, which is essential to an object-oriented solution. As discussed below, we have an integrated scheme for constraint inheritance. Furthermore they have no proposals for constraint maintenance, whereas constraints in CoLan can be selectivity retracted without recompiling any methods, and constraints can be queried as part of metadata. Instead, they implement through use of a precompiler, and presumably any change to constraints involves a rerun of the precompiler and much recompilation.

They do have interesting proposals for constraint transformation and optimisation. However, these rely on pre-existing object relationships through object identifier links, whereas our ideas of constraint inversion, described in section 6.4 are more general.

It is significant that Ode is based on a C++ data model, not on a semantic data model like that of ADAM or P/FDM which incorporates many structural constraints. In particular, P/FDM checks on object deletion using inverses enforced through the data model but Ode has to fall back on planting extra checks for deleted items to be made on every access to the class, simply because it cannot be sure what the C++ programmer may do. This is discussed further in [10] but the philosophical point is that C++ type checks are no substitute for a strong data model enforced through a central conceptual schema which is able to evolve.

3. Overview of CoLan

3.1. CoLan syntax

A typical constraint expressed in CoLan consists of two parts. The first is the quantification part, where the variables that are going to be used are given a domain and are quantified over that domain. The second part is the main part of the constraint and consists of zero, one or more predicates that should be satisfied by the instances described by the quantification part. For example, a constraint about the permissible range of ages for a postgraduate student[1] would look like the following in CoLan:

```
forall p in postgrad
  age of p>20 and age of p<45
```

3.1.1. Attribute constraints

The above constraint restricts the range of values that a slot[2] of an instance of a class can take and is called attribute constraint. Attribute constraints are conceptually, but not syntactically, the simplest constraints that are expressible in CoLan and are equivalent to the attribute-domain constraints in relational database systems [14].

3.1.2. Numerically quantified constraints

More complex constraints are the ones that relate the object that received the update message with the rest of the objects of its class (relation-aggregate constraints in [14]). For example, the constraint:

```
exist at least 3 s in staff
  such that position of s = ''security''
```

restricts the minimum number of instances of class `staff` that satisfy the predicate part of the

[1] We will use the university schema illustrated at fig. 1. oftenly throughout the paper.
[2] We shall hereafter use interchangeably the terms *slot*, applying to a frame, and *attribute* (or *instance variable*), applying to an object.

constraint. To evaluate the validity of such a constraint in the context of a certain update, we must not only search for data locally (i.e. instance variables of the recipient object), but also globally (i.e. instance variables of other instances of the same class).

Although the quantification part is mandatory, the predicate part is optional for a CoLan expression. Some CoLan expressions quantify only the number of instances in a given domain regardless of their properties. An example of such a constraint is the following:

```
exist at most 35 p in postgrad
```

which constrains the cardinality of the object warehouse of a class (class-cardinality), or:

```
exist at least 2 t in tutor of undergrad
```

which constrains the cardinality of a set attribute (slot-cardinality).

Since attributes in the relational data model can only have a simple value, no corresponding slot-cardinality constraint exists in relational databases. On the other hand, the class-cardinality constraints are known as relation-aggregate constraints [14].

3.1.3. *Class relationship constraints*

More interesting CoLan constraints relate the instances of two classes, classes not necessarily different from each other. The following example demonstrates this:

```
forall l in lecturer
  not exists s in staff
    such that phone of l=phone of s
```

The above constraint guarantees the uniqueness of the phone numbers of lecturers. It combines search among the instances of two classes, `lecturer` and `staff`. These two classes are related through an 'is-a' link (Fig. 1), so each instance of class `lecturer` is also an instance of class `staff`. Thus, an additional condition (`l<>s`) is understood.

This constraint could not be imposed by structural or key-uniqueness constraints, because the property phone is defined over class `staff`, but the constraint holds only for class `lecturer`. If phone was defined as a key for class `staff`, then phone numbers would be unique for every member of the staff and not only for lecturers. On the other hand, we cannot override the properties of phone for class `lecturer`, because it is an attribute and not a method. Furthermore, a key-attribute named phone at class `lecturer` would not ensure the uniqueness of the phone numbers of lecturers among staff, but only among lecturers. Class relationship constraints subsume key-uniqueness, relation- and inter-relation-aggregate constraints [14].

3.1.4. *Navigational constraints*

So far, we have demonstrated CoLan expressions concerning only simple attributes, i.e. attributes whose values are simple objects, like integers, strings, etc. Complex attributes are those that 'point' to other objects by storing their object identifier (henceforth OID). These complex attributes establish relationship links between classes. A complex predicate involves the comparison of complex attributes. Those constraints that involve complex predicates are called navigational, because they are defined on a class but they constrain the values that instances of other classes can have. Using function composition we can follow these chains of instances that are linked together with binary relationships. The following constraint is an example of a navigational constraint:

```
forall l in lecturer
  age of supervised_student of l<35
```

The 'host' class of the above constraint is `lecturer`, but the constraint itself does not inhibit values only for that class, but also for the class `postgrad`, because the slot `supervised_student` contains OIDs of instances of class `postgrad`. Therefore, the constraint is really about the slot `age` of the instances of class `postgrad`, but only for the ones related to instances of class `lecturer` through the relationship `supervised_student`.

Note that since `supervised_student` is a set-valued slot, there should exist a second, universally quantified variable ranging over elements of this set:

```
forall l in lecturer
  forall s in supervised_student of l
    age of s<35
```

However, this is not necessary in CoLan, since a predicate over a set implies that the predicate is applied to each element of the set, e.g.:

```
age of supervised_student of l<35≡
  forall s in supervised_student of l
    age of s<35
```

Navigational constraints are more general than either functional dependency or referential integrity constraints in relational databases [14]. More complex navigational constraints can be expressed by using complex attributes both in the quantification and the predicate parts. This use of constraints produces very complex relationships and restrictions among instances of different classes:

```
forall l in lecturer
  forall s in supervised_student of l
    research_interest of tutor of s=research_interest of l
```

The above constraint is a recursive one, because it relates class `lecturer` with itself through class `postgrad` and the complex attributes `supervised_student` of `lecturer` and `tutor` of `postgrad`. However, fully recursive constraints, e.g. transitive closure constraints, cannot currently be expressed in CoLan.

3.1.5. Constraints with mathematical operations

Finally, CoLan allows the user to express constraints with complex predicates using mathematical operations, i.e. addition, multiplication, etc. For example, one can state constraints like the following:

```
forall e in employee
  salary of e+commission of e>=10000
```

These constraints allow combinations of instance attributes or even attributes of different instances to be related. Thus they resemble tuple constraints of relational databases [14].

We have tried to demonstrate the flexibility of CoLan through examples of graded complexity. The complete syntax of CoLan in BNF notation is shown in Appendix A.

3.2. Operational semantics of CoLan expressions

CoLan expressions have clear (and obvious) declarative semantics, which are those of normal set theory with quantifiers, but complex operational (or update) semantics. For example, consider the attribute constraint:

```
forall l in lecturer
  age of l>25 and age of l<70
```

The above has obvious declarative semantics and the following operational semantics:
- If a new lecturer is created make sure that his/her age is within the range (25,70);
- An existing lecturer cannot have his age updated beyond the limits.

Thus, a single CoLan expression has consistent but differing operational semantics depending on the attempted update. This happens because CoLan implementation is based on incremental checking through atomic transactions constituted by single messages sent to individual instance objects, and not by examining database states. This approach exhibits certain features:
- The code generation task is complicated, as different pieces of code must be generated for each method;
- Constraint checking is more efficient, since constraints are only checked incrementally, taking into account the previous consistent status.

3.3. Quantifiers in CoLan

CoLan uses three kinds of quantifiers that are very important in the description of constraints. These are the universal, the existential and the numerical quantifiers (Table 1). These three categories of quantifiers are really only two, since the existential quantifiers `exists` and `not exists` can be re-stated using one of the other categories. More specifically, the `exists` quantifier is equivalent to the numerical quantifier `exist at least 1`. The `not exists` quantifier is equivalent to the `exist at most 0` numerical quantifier or

Table 1
CoLan quantifiers

name	CoLan symbol	logical symbol
universal	`forall <var>`	$\forall$ `<var>`
existential	`exists <var>`	$\exists$ `<var>`
non-existential	`not exists <var>`	$\neg\exists$ `<var>`
numerical	`exist <predicate> <number> <var>`	----

it can be transformed into the universal `forall` quantifier, provided that the predicates of the predicate part are negated, as in FOL.

The purpose of quantifiers is to define the range of instances of a class over which the restrictions of the predicate part hold. Using numerical quantifiers one can have a very fine control over the database and express very specific constraints. For example, the phrase '*Two research assistants at most should share a phone*,' is easily expressed in CoLan, as:

```
forall r in ra
  exist at most 1 r2 in ra
    such that phone of r2=phone of r
```

Range quantifiers are not yet implemented in CoLan, but the system can be easily extended to transform a constraint with a quantifier `exist at least n & at most m` into two complementary constraints, relying on the AND semantics of constraint checking.

The universal and existential quantifiers are equivalent to the quantifiers of predicate calculus, but the numerical quantifiers are not expressible in logic. However, while quantifiers in FOL can be used to express incomplete knowledge and reason about it, CoLan quantifiers are used only to restrict the possible states that a database can have after an update. For example, consider the following constraint:

```
exists e in employee
  such that position of e=''MANAGER''
```

The operational semantics of the above constraint is that the number of managers in the database should not fall below one, i.e. it means that the database should have at any time at least one specific instance with the prescribed property. The same expression in FOL just means that there exists a manager in the database, but it is not like an OODB, where we need to create an object explicitly for each manager, so that we know who they all are. Instead, FOL can reason based on incomplete information.

The use of `such that` in CoLan expressions is optional. It is mainly used as a syntactic convenience for the constraints with numerical quantifiers. For example, the following constraints are very clearly stated with the use of `such that`, while their meaning is confusing if we omit it:

-
  ```
  exist at least 3 s in staff
    such that job of s=''Security''
  ```

-
  ```
  forall p in postgrad
    exists l in lecturer
      such that research_interest of l=research_interest of p
  ```

On the other hand, simple CoLan expressions with only universal quantifiers do not need the use of `such that`:

-
  ```
  forall u in undergrad
    year of u>0 and year of u=<4
  ```

-
  ```
  forall l in lecturer
    forall s in staff
      phone of l <> phone of s
  ```

We note that a constraint with a universal quantifier can always be transformed into one with the numerical quantifier `not exists`, provided the predicate part is negated and the `such that` keyword is added:

-
  ```
  not exists u in undergrad
    such that year of u=<0 or year of u>4
  ```

-
  ```
  forall l in lecturer
    not exists s in staff
      such that phone of l=phone of s
  ```

Finally, we note an exception to the above rule of using `such that`, when a quantified variable is further restricted by a predicate (or an embedded quantified expression). In such cases, the keyword `such that` is obligatory, even with the universal quantifier:

-
  ```
  forall s in staff
    such that status of s=''honour''
      not exists s1 in students_advised of s
        such that status of s1 <> ''honour''
  ```

-
  ```
  forall s in student
    such that enrollment of s=''part time''
      year of s=<6
  ```

In the last example, `such that` is used for the restriction of the domain from students to part-time students only, while the actual constraint is about the year of study of those students. The latter is not accompanied by the `such that` keyword, since it is inside the scope of a universally quantified variable.

3.4. *Equivalencies to logic form*

In this section we demonstrate how CoLan relates to first order logic (FOL) using some simple examples. We show how to express universal implications using *forall* and *not exists*, and existential statements using *exists*. We note that some constraints such as the numerical quantifiers of Section 3.1 cannot be expressed in FOL. Finally, we compare CoLan to ALICE, a constraint description language based on FOL [30].

CoLan expressions consist of two parts, the quantification part and the predicate part. The former quantifies over variables and binds them to a domain. This domain might be a class or a set-type slot of a class. In FOL we express that a variable is bound with an entity of a domain through a unary relationship, i.e. with a predicate of arity one. For example, the CoLan expression:

```
forall l in lecturer
  P(l)
```

is translated into FOL thus:

```
∀l lecturer(l)→P(l)
```

The predicate part of CoLan includes comparisons between values of slots of the same or of different entities. This is not feasible in pure, function-free FOL, therefore we cannot directly compare the values of the slots. In most constraint languages that are based on a restricted function-free FOL, binary relationships (i.e. predicates with arity two) must be first used to "retrieve" the properties of an entity and then be compared through comparison predicates. For example, the predicate:

```
age of l>50
```

is translated into FOL as an implication [19, 20] and not as a conjunction:

```
age(a, l)→gt(a, 50)
```

because it reads in English as: '**If** *A is the age of L* **then** *A is greater than* 50'.

The only way that an implication can be proven false is for the premise to be true and the conclusion to be false. That is when the constraint is really checked and it fails. Thus the complete FOL expression for the constraint:

```
forall l in lecturer
  age of l>50
```

is the following:

```
∀l, a lecturer(l) ∧ age(a, l)→gt(a, 50)
```

More complex constraints are the ones that join different classes. These constraints require variable sharing between different binary relationships of at least two quantified variables. Consider the constraint:

```
forall l in lecturer
  not exists s in staff
    such that phone of l=phone of s
```

This translates into pure FOL:

$\forall$l, p lecturer(l) $\wedge$ phone(p, l) $\rightarrow \neg\exists$s, pl staff(s) $\wedge$ phone(pl, s) $\wedge$ eq(p, pl)

or,

$\forall$l, s, p, pl lecturer(l) $\wedge$ phone(p, l) $\wedge$ staff(s) $\wedge$ phone(pl, s) $\rightarrow$ $\neg$eq(p, pl)

If Datalog with function symbols is allowed instead of pure, function-free FOL, then the above reduces to the more concise expression:

lecturer(l) $\wedge$ staff(s) $\rightarrow \neg$eq(phone(l), phone(s))

Although these expressions are obvious to mathematicians, our experience is that naive users find them more difficult to read than the CoLan expressions. In fact we have decided not to include an explicit implication operator in CoLan, because it is more natural in ordinary language to use `not exists` following a universal quantifier. Thus, the choice between the full functional style of CoLan (with the pseudo-natural front-end), the formalism of Datalog with function symbols and the pure FOL is dictated by the intended user.

We now compare CoLan with ALICE, a "*declarative constraint language for the expression of complex, logic-based constraints in an OODB environment*" [30]. ALICE is used for expressing constraints against schemas described using CORAL semantic data model [29]. Its expressions are much like first order logic expressions and they are designed to be translated into FOL. For example consider the following constraint expressed in ALICE:

"*A member of the staff with a status of 'honour' can only advise honour students*".

```
all S in staff (where S.advisor_status=''honour'')
  implies: (all S1 in F.students_advised implies:
                (S1.status=''honour''))
```

We can see that the above expression is not easy for a naive user, for whom *implies* is hard to use and understand, compared to the following equivalent CoLan expression:

```
forall s in staff
  such that status of s=''honour''
    not exists s1 in students_advised of s
      such that status of s1 <> ''honour''
```

or even more concisely (see section 3.1.4):

```
forall s in staff
  such that status of s=''honour''
    status of students_advised of s=''honour''
```

Therefore, CoLan is more powerful than FOL and ALICE in certain aspects such as numerically quantified statements and evaluation of functions. FOL and ALICE can capture more meaning by expressing very complicated situations (using very complicated expressions), but it is not clear that the average user can comprehend this extra subtlety. Logicians will obviously prefer Datalog and ALICE formalisms, while scientists, engineers and others for whom CoLan is intended will find the functional style easier to use.

4. Background to ADAM

ADAM is an OODB management system that views data as objects and their properties as slots, just like in a frame-based system [24]. ADAM follows completely the object-oriented paradigm, concerning the use of methods and message-passing between objects as the only way of communication between the objects and the user [13]. ADAM differs from other OODBs because it handle events, rules, methods and data in a uniform approach [7]. ADAM was originally developed using Quintus Prolog and C, in Aberdeen University [13, 22].

In ADAM, objects are divided into meta-classes, classes and instances. When the system is compiled, the meta-class called `meta_class` already exists. All subsequent classes are created by sending messages to meta-classes such as `meta_class`.

The use of meta-classes in ADAM is a very flexible and powerful way for extending the facilities of ADAM system incrementally [23, 25]. To be more specific, there are meta-classes that create new classes when messages are sent to them. The ADAM system has a predefined meta-class named `meta_class` that provides the default ADAM behaviour for creating new classes. Should the user want to extend this default behaviour, then he/she can easily re-direct the messages to user defined meta-classes that override method `new` of the original `meta_class`. There are also meta-classes whose sole purpose is to create `mixin` classes that hold definitions for commonly used methods. Mixin classes do not have any instances nor any slots, but they serve as method depositories. Every new top-level user class that is created should be connected to `mixin` classes through 'is-a' links, in order to inherit these methods.

Everything in ADAM is done using the message-passing paradigm of SmallTalk [11]. The syntax of messages in ADAM is the following:

```
message⇒object
```

In response to messages sent to objects, methods are invoked which perform the operation denoted by the message. Methods can vary from simple slot retrieval or update operations to complex procedures or even programs. Method bodies in ADAM are arbitrary pieces of Prolog code.

4.1. *Expressing constraints in ADAM*

Constraints are expressed and checked in ADAM by altering the default method definitions, just as in C++ and other object-oriented languages. This approach is very awkward because one has to translate the semantics of the constraint to subsequent Prolog code and then to replace the method definitions for updates and modifications inside the classes and meta-classes of the ADAM database. It is complicated because one has to be very careful with the meta-classes that he/she alters and the inheritance problems that overriding can cause.

To demonstrate the complexity of generating constraint checking in ADAM by the user, we give the following example. Consider the university schema of Appendix B. An example of a constraint could be the following:

"*Lecturers should not share their phone with any other members of the staff*".

We need to check this constraint whenever a phone number is inserted into slot phone of an instance of class `lecturer`, and so we override the default method `put_phone` in class `lecturer` with the following call:

```
:- put_method([      % Create a new method for inserting values in slot phone of lecturer
    % Method name(visibility, I/O argument types, I/O argument variables)
    (put_phone(global, [], [integer], [], [Phone]):-
      % Are there any instances with the same phone number in class staff?
      (get_by_phone([Phone],_)⇒staff→
          % If yes the constraint is violated! Print a message and fail!
        (write('There cannot be two lecturers
                 with the same phone number!'),
       nl, fail);
          % Otherwise evoke the default method at the superclass of the current class
        put_phone([Phone])⇒super))
    ])⇒lecturer.
```

However, this is not the only piece of code to create to ensure the consistency of the database. Methods `update_phone` and `new` should also be altered to exclude every potential modification of the database that would lead to an inconsistency. Code should be generated for the corresponding methods of class `staff`, too, to inhibit a `staff` instance from violating the constraint. A single constraint here affects at least six methods.

The case of extra constraints on the same method is even more complicated. Consider for example the constraint that lecturer phones have numbers that begin with 2. The piece of code that ensures this for method `put_phone` of `lecturer` is the following:

```
:- put_method([
   (put_phone(global, [], [integer], [Phone]) :-
     ((Phone<2000; Phone>=3000)→% Is the phone number outside the range?
       (write('Phone numbers of lecturers must begin with 2!'),
       nl, fail);       % If yes the constraint is violated! Print a message and fail!
       put_phone([Phone])⇒super))  % Otherwise evoke the method at superclass
   ])⇒lecturer.
```

However, should the above goal is executed, then the previous code for enforcing the unique-phone constraint is lost/overridden. In order to enforce both constraints, one has to retrieve the old method body, inject the apropriate piece of code to the appropriate place and replace the old method with the new one. But the latter is complicated and error-prone, even for the database designer.

Obviously ADAM needs a user-friendly high-level constraint specification language with an efficient implementation, because otherwise constraints are expressed in a Prolog program whose semantics are hidden between the lines of a method description (which is a major problem for object databases [27]). Hence, constraints are not so clearly stated for the rest of the users apart from the programmer.

5. CoLan structure

5.1. Design philosophy

There are two conflicting arguments about the implementation of constraints. The former [8] says that constraints should be separate objects with their own structure, and the class descriptor should only refer to them through their object identifiers. The latter says that the former is too slow when implemented, so it handles constraints as meta-data of the class descriptor and it refers to them using their code translation. The latter is easier to implement and it performs better but the former is more general and flexible as it allows more control of constraint behaviour.

An intermediate approach is to handle constraints as distinct objects for maintenance purposes but to *cache* code that ensures constraint checking inside the class to which it applies. To be more specific, when a CoLan expression is inserted in the schema, its high level description along with other properties of the constraint are stored as separate objects, as instances of class `constraint`. The generated code is stored within the class it applies to and is retrieved by calling a special predicate at run-time.

The main difference between our system and a system such as ABEL [8] is that in ABEL[3] the constraint expression is translated into an active rule [7, 9] and the rule is stored and treated as a separate object, not the constraint itself, while in CoLan constraints are objects but only from the system's point of view. Thus the user in CoLan sees the constraint just as a simple property of a class declaration.

[3] ABEL is an extension of ADAM with active rules, relationships, versioning, constraints, etc.

5.2. Constraint inheritance and checking

Constraints in an OODB are integrity rules that hold between data and inhibit some transactions in the database. We believe that constraints defined on a class should be inherited by all its subclasses without any option to override them with a more specific version or even cancel them completely. As Brachman says in his critique of frames [3]: "*Cancellation inheritance is disastrous*". Thus constraints should be like definitional properties attached to classes and therefore must not be overridden [8, 25]. In this respect the inheritance strategy of constraints in CoLan has to be completely different from that of methods, which can be specialised and/or completely overridden.

In this paper we assume that there are no exceptions to this rule, i.e. every object should satisfy all inherited constraints, and hopefully no contradictions occur. In cases where constraint contradictions, overriding etc., are allowed, then these must either be declared explicitly by the database designer [2] or else inferred by a logic-based constraint analysis tool [4].

In ADAM, inheritance of a default method definition is achieved by delegating the message received by an instance of a class to its superclass. However, the database designer must remember to include this call to the superclass when selectively re-defining part of a method, otherwise the default behaviour will be lost/overridden.

To overcome this in CoLan, the system ensures inheritance and checking of all the constraints, using a special predicate called `check_constraints`, which cannot be bypassed (see Appendix E). When an update message is received by an instance of a class, a call to this predicate is made. This predicate is responsible for inheriting and checking constraints defined in that class or its superclasses consecutively. Inheritance proceeds in a top-down fashion, i.e. first constraints defined at the highest superclass of the target class are checked and then constraints defined in the immediately lower level, etc., until constraints defined at the target class are checked. Actually, what is inherited is not the high level description of the constraint but the appropriate piece of generated code that ensures the validity of the corresponding constraint. When the piece of code is obtained, either from the target class or from a superclass, it is applied in the context of the current update and the recipient object.

If the piece of Prolog code succeeds, i.e. the update does not violate the constraint, then the algorithm backtracks to get another piece of code that corresponds to another constraint. If all constraints are inherited and checked successfully, the update is accepted and the default method that responds to the update message is executed. If one of the constraints is violated, then no further constraints should be checked, because the update is invalid anyway. In this case the update is rejected and the default method is not invoked. An informative error message is displayed and the original call fails.

5.3. Constraint storage and maintenance

Constraints may be inserted and/or deleted in the schema at any time in the life cycle of the database and not only during the initialisation stage. The only problem associated with constraint insertion is that currently there is no initial state checking, when the constraint first enters the database, therefore there is no guarantee that the database already satisfies the new constraint, but this extension is straightforward. Meanwhile, we have adopted a temporary solution that facilitates incremental loading of data (see Section 6.2.3).

Constraint insertion is realised by sending a message put_constraint to the class to which the constraint applies. The CoLan expression is parsed, the parse tree is optimised and is fed to the code generator. The generated pieces of code affect certain methods of the class and are attached to the class. For example, the following constraint:

```
forall p in person
  age of p>0 and age of p<120
```

generates code for slot age and more specifically for methods put_age and update_age, which might violate the constraint. The code fragments for the above two methods are stored inside the slots (or class variables) put_constraints and update_constraints of class person, respectively.

Thus, if an update is applied to a slot which has no constraints then there is a slight overhead to pay in checking for absence of relevant constraints in the constraint list in these class variables. This check in fact could be made at constraint installation time. The significant point is that we do not waste time running code fragments which do not apply to the slot in question.

Meanwhile, an instance of class constraint is created that keeps the high-level description of the constraint and the links to the actual produced code. The pieces of code do not lose their identities after their generation, because they keep their links with the constraint that generated them. As constraints are objects, they have distinct identifiers like every other object in the system. This identifier is stored within every piece of code generated by the constraint.

The rationale for such a complex relationship between constraints and classes they apply to, is revealed by constraint deletion (see Example 4 in Appendix C). When the user wants to delete a constraint, he/she sends a delete_constraint message to the class the constraint belongs to. If the constraint does belong to the recipient class, then CoLan finds its identifier and deletes every piece of code that relates to the constraint, using the identifier attached to those pieces of code. The deletion of code includes not only the recipient class, but also every other class that is involved in the constraint. The latter are stored in the slot classes of the constraint-object. Finally, the actual constraint-object is deleted from the database.

A simple extension to CoLan could *inhibit* constraint checking of selected constraints without deleting them, simply by changing the value of a switching slot in the constraint-objects and temporarily disabling them. This requires though that a constraint check on existing data is performed immediately after re-enabling a temporarily disabled constraint. Constraints can also be replaced by other constraints, but this also needs an initial state check as mentioned earlier.

In Appendix E we include details about how the method invocation approach to constraint checking has been accomplished in ADAM, using the powerful concept of meta-classes. In fact, any OODB system that supports meta-classes can incorporate more or less the CoLan constraint checking sub-system.

6. Constraint compilation and code generation for CoLan

This section discusses the generation of actual constraint enforcement code from CoLan specifications. In Appendix C we have included an extended script of interaction with CoLan, which demonstrates the flexibility, friendliness and usefulness of both the language and its implementation. Appendix D gives some examples of code fragments generated by the constraint compiler.

6.1. Performance considerations

The code-generation time for a constraint is about 5 seconds on a SUN3 workstation under Quintus Prolog v.2.5 and ADAM v.2.2 [1]. However, this time is not very important, since constraints are created only once and the produced code is directly used at run-time. Note that the insertion of constraints to the schema assumes that the constraint is satisfied by existing data. Once the constraint is translated into the appropriate pieces of Prolog code and the appropriate methods are altered, the database cannot fall into an inconsistent state because each update that could violate a constraint is 'guarded' by one or more pieces of code that check its validity.

This is one of the major performance benefits of using method invocation to check constraints. Constraints are checked only when needed and in the appropriate context. Only applicable constraints are checked, because code is cached and therefore indexed according to the class the constraint refers to. A similar approach to checking only applicable constraints is used in most triggered-based systems [5, 15]. However, these systems are based on the relational model, and thus several fundamental differences can be found. Specifically, in relational systems only triggers {INS, DEL} are used, while in OODB systems every recognisable method name constitutes a trigger [7]. This complicates matters in CoLan, compared to constraint checking systems for relational databases.

In contrast to a constraint checking sub-system that relies on an active rule interpreter [7], the method invocation approach does not affect those parts and methods of the database that do not have to do with constraints. These, for example, include all the retrieval methods (e.g. `get`, `get_age`, etc.). The latter play an important role in query answering, therefore CoLan does not affect the query evaluation speed of ADAM.

On the other hand, an active rule interpreter has been reported [7] to generally degrade ADAM's performance by a factor of two. This is a consequence of the fact that an active rule mechanism must be incorporated in the message-sending and method-execution mechanism, as a side-effect of method invocation. Therefore, even if a message does not constitute a detectable event, the detour from default method execution to check whether the message is an event or not, slows down the system. Thus in terms of performance, active rules are not a very good solution for constraint checking in ADAM, unless they are used anyway to provide features other than constraint checking [25].

A design aim is to improve performance of CoLan wherever possible, mainly at run-time. Parse tree transformation techniques to optimise generated code are described later. However, the main performance improvement comes through the avoidance of global computations where local checks could suffice to accept or reject an update.

6.2. Code generation

According to the formal syntax of CoLan (Appendix A), a general CoLan expression is defined as:

```
Q₁ var₁ in ent₁ [such that expr₁(var₁)]
  ....................
  Qᵢ varᵢ in entᵢ [such that exprᵢ(var₁, . . . , varᵢ)]
    ..................
    Qₙ varₙ in entₙ [such that exprₙ(var₁, . . . , varₙ)]
  [[such that] c(var₁, var₂, . . . , varₙ)]
```

where i ranges from 1 to n and:

- Q_i is a quantifier, either the universal `forall`, or a numerical `exist rel_op N`, where `rel_op` is one of the following: >, >=, <, =<, =, \=, and N is a non-negative integer. Note that `not exists` means `exist=<0` (see Section 3.3).
- var_i is a variable that ranges over the entity ent_i.
- ent_i are entities and they might either be named classes of the OODB schema, or else subsets of a class computed by functions (simple or composed) of variables that have been declared earlier:

  ```
  entᵢ=classᵢ
  entᵢ=function(varⱼ), j<i
  ```

  An exception occurs for i = 1. The ent_1 entity can either be a class or a function (simple or composed) of a class, since there are no previously defined variables. However, when the parse tree optimiser detects a parse tree with ent_1 bound to a function of a class it transforms it (see Section 6.3) to a new expression with an extra universally quantified variable at the outer loop, ranged over the class. The original 'first' variable now ranges over the function of the new variable. Thus, we can safely assume that ent_1 is always bound to a class.
- $expr_i(var_i)$ are optional CoLan sub-expressions, that restrict the values that the variables var_i can take over the domain ent_i. If they are missing, they are considered as tautologies, i.e. true under each interpretation. The expressions $expr_i$ may be simple predicates, i.e. comparisons applied to constants or functions of variables (see last example in Section 3.4), or else they can be embedded quantified expressions (see inverted constraint example in Section 6.4.1):

  ```
  exprᵢ(var₁:ent₁, . . . , varᵢ:entᵢ)=simple_predᵢ(var₁, . . . , varᵢ)
  exprᵢ(var₁:ent₁, . . . , varᵢ:entᵢ)=
      ᵉQᵢ ᵉvarᵢ in ᵉentᵢ
        such that ᵉsimple_predᵢ(var₁, . . . , varᵢ, ᵉvarᵢ)
  ```

  where the index 'e' identifies the embedded expression and the notation `x:type` is used to permit type-checking of variable x over type type. Simple predicates are conjunctions or disjunctions of comparisons among simple or composed functions of variables and/or constants:

$$\texttt{simple\_pred}_i(\texttt{var}_1, \ldots, \texttt{var}_i) = \texttt{simple\_predicate}_i(\texttt{var}_1, \ldots, \texttt{var}_i)$$
$$\texttt{simple\_pred}_i(\texttt{var}_1, \ldots, \texttt{var}_i) = \texttt{simple\_predicate}_i(\texttt{var}_1, \ldots, \texttt{var}_i) \; \{\texttt{and}|\texttt{or}\} \; \texttt{simple\_pred}_i(\texttt{var}_1, \ldots, \texttt{var}_i)$$
$$\texttt{simple\_predicate}_i(\texttt{var}_1, \ldots, \texttt{var}_i) = \texttt{operand}_1(\texttt{var}_1, \ldots, \texttt{var}_i) \; \texttt{rel\_op} \; \texttt{operand}_2(\texttt{var}_1, \ldots, \texttt{var}_i)$$
$$\texttt{operand}_j(\texttt{var}_1, \ldots, \texttt{var}_i) = \texttt{constant}$$
$$\texttt{operand}_j(\texttt{var}_1, \ldots, \texttt{var}_i) = \exists k \le i, \; \texttt{func}(\texttt{var}_k)$$

where $j \in \{1, 2\}$, constant is a number or string and func(x) is a simple or composed function of variable x.

- c is a simple predicate representing a constraint, and not an embedded quantified expression. If c is missing, then it is considered a tautology. Note that the use of such that is optional, since it is just a syntactic convenience that follows the last quantified variable with a numerical quantifier, but not with the universal quantifier.

Now, we define a series of predicates $\texttt{pred}_i$ in order to denote the meaning of a syntactically correct CoLan expression:

$$\texttt{pred}_i(\texttt{var}_1{:}\texttt{ent}_1, \ldots, \texttt{var}_{i-1}{:}\texttt{ent}_{i-1}) = \texttt{Qpred}(Q_i, S_i(\texttt{var}_1, \ldots, \texttt{var}_{i-1})), \; i \le n \qquad (1)$$
$$\texttt{pred}_{n+1}(\texttt{var}_1{:}\texttt{ent}_1, \ldots, \texttt{var}_n{:}\texttt{ent}_n) = c(\texttt{var}_1, \ldots, \texttt{var}_n) \qquad (1a)$$

where:

- $\texttt{pred}_i$ is a predicate that denotes whether the nested constraint that begins from variable $\texttt{var}_i$ is satisfied by instances of i-1 entities that appear before $\texttt{ent}_i$ in the CoLan expression. $\texttt{pred}_i$ contains i-1 free variables.
- S_i is a ZF expression [28], i.e. a bag of boolean values, that correspond to the satisfaction (or not) by each of the instances of the class $\texttt{ent}_i$ of all the restrictions introduced after the i-th variable. S_i also contains i-1 free variables.

$$S_i(\texttt{var}_1{:}\texttt{ent}_1, \ldots, \texttt{var}_{i-1}{:}\texttt{ent}_{i-1}) = \{\langle \texttt{pred}_{i+1}(\texttt{var}_1, \ldots, \texttt{var}_i)\rangle \mid \texttt{var}_i \leftarrow \texttt{ent}_i; \; \texttt{var}_i \ne \texttt{var}_1; \; \ldots; \; \texttt{var}_i \ne \texttt{var}_{i-1}; \; p_i(\texttt{var}_1, \ldots, \texttt{var}_i)\} \qquad (2)$$

- The inequalities in expression (2) are needed to check that the instance denoted by variable $\texttt{var}_i$ is not a specialisation of or identical to any of the objects from $\texttt{var}_1$ to $\texttt{var}_{i-1}$ and thus to avoid comparing an object with itself (see Section 3.1.3).
- The notation ⟨P(...)⟩ stands for the boolean value of the predicate P under the interpretation of a given tuple of argument values.
- p_i is a predicate that corresponds to the restrictions placed on the instances of entity $\texttt{ent}_i$,

through the expression $\texttt{expr}_i$. As $\texttt{expr}_i$ can either be a simple predicate or an embedded expression, $\texttt{p}_i$ is defined as follows:

$$\texttt{p}_i(\texttt{var}_1\texttt{:ent}_1, \ldots, \texttt{var}_i\texttt{:ent}_i)=\texttt{simple\_p}_i(\texttt{var}_1, \ldots, \texttt{var}_i)$$
$$\texttt{p}_i(\texttt{var}_1\texttt{:ent}_1, \ldots, \texttt{var}_i\texttt{:ent}_i)=\texttt{Qpred}({}^e\texttt{Q}_i, {}^e\texttt{S}_i(\texttt{var}_1, \ldots, \texttt{var}_i))$$

where ${}^e\texttt{S}_i$ is the ZF expression that corresponds to the embedded quantified expression:

$${}^e\texttt{S}_i(\texttt{var}_1\texttt{:ent}_1, \ldots, \texttt{var}_i\texttt{:ent}_i)= \{\langle {}^e\texttt{simple\_p}_i(\texttt{var}_1, \ldots, \texttt{var}_i, {}^e\texttt{var}_i)\rangle \mid {}^e\texttt{var}_i \leftarrow {}^e\texttt{ent}_i\}$$

- The predicate Qpred checks the cardinality of a ZF expression S, according to the quantifier Q:

  1. If Q=''forall'', then Qpred(Q, S)=(#FALSE(S)=0)
  2. If Q=''exist rel_op N'', then Qpred(Q, S)=(#TRUE(S) rel_op N)

The notation #ST denotes the cardinality of the set (or bag) ST. Functions FALSE and TRUE are themselves defined as ZF expressions:

```
TRUE(S)={x|x←S; x=true}
FALSE(S)={x|x←S; x=false}
```

We may notice that for i = 1, equations (1) and (2) are re-written as:

$$\texttt{pred}_1=\texttt{Qpred}(\texttt{Q}_1, \texttt{S}_1) \tag{1b}$$
$$\texttt{S}_1=\{\langle \texttt{pred}_2(\texttt{var}_1)\rangle \mid \texttt{var}_1 \leftarrow \texttt{ent}_1;\ \texttt{p}_1(\texttt{var}_1)\} \tag{2b}$$

In order to evaluate $\texttt{pred}_1$, $\texttt{S}_1$ must be evaluated over all instances of $\texttt{ent}_1$. However, assuming the database state satisfies $\texttt{pred}_1$ before an update, then only the value of $\texttt{pred}_2(\texttt{var}_1)$, where $\texttt{var}_1$ is the message recipient object, will be added to $\texttt{S}_1$ by the update. If $\texttt{Q}_1$ is ''forall'' then we can safely ignore all $\texttt{var}_1$'s in $\texttt{S}_1$ apart from the current (incremental checking). On the other hand, if $\texttt{Q}_1$ is a numerical quantifier, then $\texttt{S}_1$ must be re-calculated unless counter optimisation is used (see Section 6.2.2).

6.2.1. *Methods affected by constraints*

The expressions (1) and (2) are an intermediate step to Prolog code fragment generation. Actually, several pieces of code are produced, one for each different function whose update could possibly violate the constraint. Specifically, the code generator produces code for each function in $\texttt{pred}_1$ that is directly attached to class $\texttt{ent}_1$. In order to determine which these are, the generator scans the ZF-expressions and collects all simple functions of $\texttt{var}_1$, looking inside $\texttt{expr}_i$'s, $\texttt{ent}_i$'s and c. If there are composed functions of $\texttt{var}_1$ the generator will isolate the inner-most function. Furthermore, methods new and/or delete are also affected by the constraint. This technique produces code only for class $\texttt{ent}_1$. However, methods of other classes, that are related to $\texttt{ent}_1$ through the constraint, must also be dealt with. This is taken care of by constraint inversion.

Quantifiers also affect the number and nature of methods that could possibly violate a constraint. An example will make this statement more clear:

```
exists e in employee
  such that position of e=''MANAGER''
```

The above constraint insists that at least one instance of class `employee` should be a manager. The constraint does not "care" if more than one managers exist, therefore methods that can only introduce more managers, like `new`, `put_position`, etc., could not possibly violate it. The constraint needs code only for methods that could reduce the number of managers in the database, like `delete`, `delete_position`, etc. CoLan system avoids producing unnecessary code by combining the operational semantics of the quantifier and the affected method (Table 2). In the column "semantics" the two words "new" and "delete" denote the following types of methods:

- `new`, `put_⟨slot⟩` and `update_⟨slot⟩` for "new" semantics, and
- `delete`, `delete_⟨slot⟩` and `update_⟨slot⟩` for "delete" semantics.

Some CoLan expressions may exhibit both the above properties, i.e. they may be violated both by "delete" and "new" type of methods. These for example are the ones that have the `exist exactly` quantifier:

```
exist exactly 1 s in staff
  such that position of s=''Head of Department''
```

In the above example, both a deletion and a creation of an instance that satisfies the predicate part would cause a violation. Therefore, code must be generated for all the methods that alter: a) the slot `position` of an instance of class `staff`, or b) the warehouse of class `staff`.

Table 2
Operational semantics of CoLan quantifiers

Quantifier	**Semantics**
forall	new
at most, no more than, =<	new
less than, <	new
at least, no less than, >=	delete
more than, >	delete
exactly, only, =	new, delete

6.2.2. Code generation for one-variable constraints

We will now demonstrate how code is generated based on $\mathtt{pred}_i$'s and S_i's defined above. We will only consider CoLan expressions with one and two variables, since the code for the general case would look very incomprehensible and is not currently implemented.

We first consider a one-variable CoLan expression (syntax discussed in Section 3.3):

```
Q₁ var₁ in class₁
  [such that] c(var₁)
```

The predicate $\mathtt{pred}_1$ for such an expression is given by (1b). If the quantifier Q_1 is the universal one, then the code generated for the above constraint is:

```
((get_values(c,Var₁',Values'),
  c(Values'))→true;
              (print_error, fail))
```

where $\mathtt{Var}_1'$ is bound to the message recipient object, *get_values*(*c*, *Var*$_1$', *Values*') is a macro for a piece of code that retrieves in *Values'* those slot values of the instance $\mathtt{Var}_1'$ of $\mathtt{class}_1$, to be used in predicate c and *print_error* is a system predicate that informs the user about constraint violation. A concrete example for the above case is given in Appendix D – Example 1. Note that we do not iterate over instances of $\mathtt{class}_1$ since a check of the constraint in the context of the message recipient object $\mathtt{Var}_1'$ suffices. Thus, the ZF-expression S_1 is not constructed at all.

The source of the slot values returned by the macro *get_values* may vary and it depends on the actual method for which the code fragment is generated. More specifically, some of these values are available via retrieval methods of instance variables, while some others are available through the parameters of the method. Since several methods are affected by a single constraint, the sources of these values differ for each code fragment. Example 5 in Appendix D demonstrates how the same constraint affects two methods and how the values for the predicate are obtained via the method parameters or the instance variables of the message recipient object.

If the quantifier Q_1 is numerical, then the code generated for the one-variable constraint is:

```
((get_values(c,Var₁',Values'),
  c(Values'))→
          (findall(Var₁, (get(Var₁)⇒class₁,
                          get_values(c,Var₁,Values),
                          c(Values)),
                     TrueS₁),
           length(TrueS₁,Length),
           check_length(Q₁,Length))→true;
                                  (print_error, fail));
          true)
```

It is obvious that instead of the bag S_1 we construct the set TRUE(S_1) as a list of OIDs of $\texttt{class}_1$ using `findall`. Then the length of this list is checked against the numerical quantifier Q_1. Notice that before going into the `findall` loop we check if the instance $\texttt{Var}_1'$ of the current update is relevant to the constraint. If not, then the constraint is trivially satisfied (see Example 2 in Appendix D).

Using `findall` consistently provides simplicity and uniformity, because every constraint has a similar structure, however, this may prove inefficient, since for constraints with a small upper limit in the quantification part one can save time by only looking for this many instances. Thus, in future we may sacrifice uniformity and substitute the exhaustive search with other predicates for efficiency. What we will gain though is minimising average case complexity, because the worst case complexity will still remain O(n).

Another possible optimisation to avoid re-computation is to store counters of instances that already satisfy the constraint. The stored counter will be checked when an update is attempted on an instance that matches the predicates of the constraint. If the updated value of the counter would violate the predicate of the numerical quantification, then the update is rejected, otherwise it is accepted. This technique has been used in the P/FDM implementation of CoLan [10], to improve performance. However, this technique requires extra code to update the counters, even for methods that could not violate the constraint. Furthermore, its use is limited only to one-variable constraints. To demonstrate this, consider the following constraint:

```
forall r in ra
  exist at most 2 r1 in ra
    such that phone of r=phone of r1
```

In order to enhance the performance of checking the above constraint, no simple counter can be introduced, because the number of research assistants that share a phone is different for each distinct phone number. Thus there must exist as many counters as distinct phone numbers and we have a space versus time tradeoff.

6.2.3. Incremental satisfaction of numerically quantified constraints

There is a problem with constraints requiring the existence of an exact or minimum number of objects satisfying certain conditions. Clearly such constraints cannot be satisfied in an empty database. We could deal with the problem by only allowing such constraints to be added once the database is in a state to satisfy them. We decided instead to take the approach of allowing the database to be inconsistent with such constraints, but to print a warning. Where an update moves the number of objects closer to the exact number or the minimum threshold then we allow it (with a warning). Otherwise we disallow it. Once the database does satisfy the constraint then we enforce it strongly with no exceptions. This is a pragmatic convenience when loading data incrementally, but does not affect the principles of CoLan. Its effects are obvious and well-defined for constraints with numeric quantifiers.

For example, consider the constraint:

```
exist at least 2 s in staff
  such that position of s=''Secretary''
```

The insertion of the above constraint into an empty database would just produce a warning and the constraint is accepted. When the database is populated by inserting two secretaries. CoLan does not complain. Once the lower limit of two secretaries is reached, though, CoLan would not allow the reduction of the number of secretaries below two. The same considerations apply to the `exist exactly` quantifier.

6.2.4. Code generation for two-variable constraints

When there are two variables in a CoLan expression, then the code generator is confronted with four cases corresponding to all the combinations of universal and numerical quantifiers for the two variables. Here we will only consider the case where the first variable is universally quantified and the second is numerically quantified (Example 3 in Appendix D or last example in Section 3.4; syntax is discussed in Section 3.3):

```
forall var₁ in class₁
  [such that expr₁(var₁)]
    Q₂ var₂ in ent₂
      such that c(var₁, var₂)
```

The code for the predicate $\mathtt{pred_1}$ of the above constraint is:

```
((get_values(p₁, Var₁', Values'),
  p₁(Values'),
  pred₂(Var₁'))→true;
                    (print_error, fail))          (3)
```

Note that predicate $\mathtt{pred_1}$ includes a nested call to predicate $\mathtt{pred_2}$ which corresponds to the following code fragment:

```
(findall(Var₂, (get_entity(ent₂, Var₂),
                Var₂\=Var₁,
                get_values(c, Var₁', Var₂, Values),
                c(Values)),
          TrueS₂),
length(TrueS₂, Length),
check_length(Q₂, Length))→true;
                            (print_error, fail))      (4)
```

where *get_entity*(ent_2, Var_2) is also a macro that returns in variable `Var`$_2$ the OID of the actual instance of entity `ent`$_2$. If `ent`$_2$ is a named class, then this macro is nothing more than the call: `get(Var`$_2$`)`$\Rightarrow$`class`$_2$, which iterates over the instances of `class`$_2$. On the other hand, if `ent`$_2$ is a function of variable `var`$_1$, then the macro retrieves the appropriate OID through chained function calls. Examples 3 and 4 in Appendix D demonstrate the difference in the actual generated code.

For more than two variables, the generated code consists of a series of nested calls, as expressions (1)–(4) imply. However, there are other problems associated with constraints with more than two variables, like the lack of transactions (see Section 7.3), that led us not to include such constraints in this first implementation of CoLan.

6.3. Re-write rules

CoLan makes extended use of re-write rules to provide its full functionality. This is because the code generator of CoLan is designed to accept a limited number of parse tree patterns and to output a basic set of code fragments that correspond to the parse tree. This makes code generator simple to program, debug and extend. Rewrite rules are production rules of the form:

''*If* ⟨*Pattern*⟩ *then* ⟨*Transformed-Pattern*⟩''

The ⟨*Pattern*⟩ is a parse tree template that reflects a general type of constraint to be re-written. The ⟨*Transformed-Pattern*⟩ is the new parse tree that is returned from the re-write rule. In order to transform the former to the latter some actions should be performed, that are specific to the nature of the transformation.

An example of the use of re-write rules in CoLam is transformations between equivalent CoLan expressions whose translation to Prolog is more efficient than the original one, i.e. optimisation. The transformation is accomplished using re-write rules. A very simple re-write rule used to model a logical equivalence of CoLan expressions, is the following:

If *there are two variables* **and** *both are universally quantified*
then *change the quantifier of second variable to a* ''*not exists*'' *one* **and**
negate all the predicates of the predicate part

The above re-write rule follows from the optimisation technique that it is more efficient to check for a single violation of the rule than an exhaustive check that all instances satisfy it.

6.4. Constraint inversion

Constraint inversion is a way to rewrite a constraint so that it can be checked efficiently for an update on the state of a class member other than that in the first quantifier. In Ode [16] the constraint is copied verbatim into other classes and attempts are then made to transform it for greater efficiency. Below we describe our approach to the same problem and the interesting research issues that it raises. Constraint inversion will be better demonstrated using an example. Consider the familiar constraint:

```
forall l in lecturer
  not exists s in staff
    such that phone of l=phone of s
```

As we can see in Fig. 1, class `staff` is a superclass of class `lecturer`, so code generated for the latter is not going to be inherited by the former. Thus, if a member of the staff is created having the same phone number as a pre-existing lecturer, then because checking code is stored only in class `lecturer`, the update is going to be accepted! Consistency of the database requires code to guard class `staff`, too. CoLan system ensures this by generating a related (inverted) constraint to attach to class `staff`.

Constraint inversion is based on the operational semantics of constraints. The code generated for the above constraint looks as if it was created for the following CoLan expression:

```
forall s in staff
  not exists l in lecturer
    such that phone of s=phone of l
```

Since the `not exists` quantifier relates to the universal one, in this case inversion is simply the swap of positions of universal quantifiers, which is allowed by FOL. However it is not always easy to find CoLan expressions that would generate the appropriate code for the inverted constraint. Constraints that involve existentially or numerically quantified variables can be inverted, but the declarative semantics of the corresponding CoLan expression are not so easily stated, because the positions of an existential and a universal quantifiers are not interchangeable, as the following example demonstrates:

```
forall l in lecturer
  exists s in staff
```

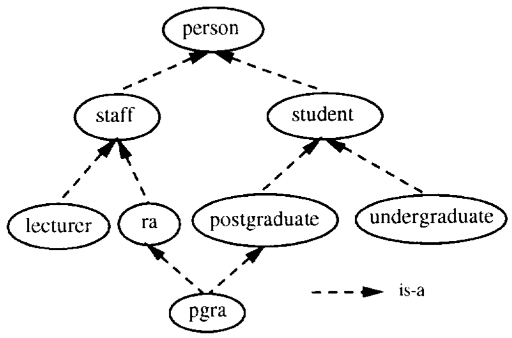


Fig. 1. University database schema.

```
    such that phone of l=phone of s
```

This constraint means that each lecturer shares his/her phone with at least one member of the staff. The opposite is not always true, i.e. a member of the staff may or may not share his/her phone with a lecturer. Thus, the following constraint is not the correct inverse form of the original constraint:

```
forall s in staff
  exists l in lecturer
    such that phone of s=phone of l
```

What we are looking for, is a CoLan expression that has the following operational semantics: "***If** a member of the staff shares the same phone with a lecturer, **and** there is no other member of the staff with the same property, **then** that member of the staff cannot change his/her phone.*" This operational semantics can be achieved by the following algorithm:

When a deletion of a member of the staff is attempted, then:

1. Check if the member of the staff shares phone with a lecturer.
2. If no, then succeed and accept the deletion.
3. If yes, then check if there exists another member of the staff with the same phone number.
4. If yes, then succeed and accept the deletion.
5. Otherwise fail and reject the deletion, because the lecturer will not share phone with a member of the staff after the deletion, as the original constraint claims.

Similar algorithms hold for phone deletion and update. However, it is not easy to find a CoLan expression to reflect the above operational semantics, so the operational semantics must be encoded directly in Prolog and stored with class `staff`. The inverted form is still bound to the original constraint, since the latter is much more simple to read and understand.

6.4.1. Limitations on constraint inversion

Constraint inversion is a difficult task that made us refrain from implementing full CoLan expressiveness. For example, constraints with two variables are restricted to expressions with their first variable universally quantified, because inversion of constraints with the first variable numerically or existentially quantified introduces certain theoretical and/or implementation problems. For example, consider the following constraint:

```
exists s in staff
  such that forall l in lecturer
    phone of s=phone of l
```

which insists that at least one member of the staff has the same phone with all the lecturers and also implies that all the lecturers have the same phone. The nearest we can get with inversion is something like:

```
forall l in lecturer
  such that not exists l1 in lecturer
```

```
    such that phone of l1=phone of l
  exists s in staff
    such that phone of s=phone of l
```

Alas, the above expression does not ensure the original constraint, because a new lecturer with a different phone number from the rest can be inserted, provided there is one member of the staff with the same phone number!

The complexity of constraint inversion, even for only two variables, led us not to include such constraints in the abilities of this first CoLan implementation, in order to first fully research the rules for implementing them. We are not implying that the inversion problem is unsolvable, but that it is quite complicated and needs careful examination. For more than two variables the problem is much harder, since all variables must be moved to the beginning of the constraint, bearing in mind that universal quantifiers can freely interchange positions, while numerical quantifiers cannot without altering the structure of the constraint.

7. Future developments of CoLan

Some features of CoLan require further research, such as: (a) Expressing dynamic constraints; (b) Using relational operator semantics for optimisation; (c) General cases of constraint inversion, etc. However their implementation does not seem to require major improvements of CoLan. For example, we will give some hints on how the first two of the above features could be incorporated in CoLan, in Subsections 7.1 and 7.2.

On the other hand, there are some deficiencies of the CoLan system that have mainly to do with the underlying system, namely ADAM. In subsections following 7.2 we will try to outline these points.

7.1. Expressing dynamic constraints in CoLan

The incremental nature of constraint checking in CoLan, leaves room for expressing and implementing dynamic constraints. Consider the following CoLan expression:

```
forall p in person
  new of age of p>=age of p
```

In the above expression a function `new` is applied to the `age` of a `person` that results in a different value for `age`, than the one mentioned on the right-hand side of the predicate expression. This is the new value introduced to the database during an update. Since constraints in CoLan are of preventive nature, i.e. they are checked only before an atomic update takes place, then it is viable to distinguish between the value introduced by the update-message and the value already stored in the slot `age` of the recipient instance. If constraints were checked after the update then old values would be lost and state transition constraints could not be checked, unless differential files are used.

The declarative semantics of the above expression is: "*Age of persons never decreases*" and the operational semantics reads:

If *an update on the age of a person is attempted,*
then *the new age should be greater or equal to the old age.*

The new function is necessary for the parser to distinguish between the old and the new values and for the user to specify them. Other expressions that involve comparison between two different slots of the same instance do not need an explicit new function, since the constraint will in turn apply to both the new and the old values:

```
forall e in employee
  salary of e>commission of e
```

The above constraint (which is enforceable in the current CoLan implementation), with respect to the new function, is equivalent to the following two constraints:

```
forall e in employee
  new of salary of e>commission of e
forall e in employee
  salary of e>new of commission of e
```

The current implementation does not include dynamic constraints, but it seems there is no problem in introducing the new function in simple cases, due to the incremental nature of constraint checking in CoLan.

7.2. *Transitive properties of relational operators*

One way to avoid global constraint checking is by exploiting the transitive properties of the relational operators >, <, etc. The transitive property of a relational operator (e.g. >) can be expressed as follows:

```
(a>b) ∧ (b>c)⇒(a>c)
```

The following example will illustrate how the above property can be used to save time of constraint checking by avoiding global computations:

```
exist less than 5 m in manager
  such that salary of m>10000
```

If the following ADAM update is entered:

```
update_salary([15000, 20000])⇒13#manager.
```

we can avoid checking the constraint globally, if we notice that the new value 20000 is greater than the old one 15000, which is greater than the lower limit of the constraint (10000). Therefore, the constraint was already satisfied and there is no need the re-check the database for all the managers with salaries greater than 10000, since this manager was already one of them (15000 > 10000) and the update does not try to add one more more-than-10000 manager.

In the current implementation of CoLan the transitive properties of relational operators are not considered and there is scope for optimisation as just illustrated.

7.3. *Lack of transactions*

There are problems with constraints about relationships among instances of the same or different classes. For example, the following constraint:

```
forall c in company
  forall d in department
    exists s in supplies
      such that c=company of s and d=department of s
```

is the CoLan equivalent of the following constraint, in English:

"*Every company supplies every department with at least one item*".

The problem of implementing such a constraint is that the existence of an instance at each one of the three classes `company`, `department` and `supplies` depends on the existence of an instance at both the other classes. For example, inserting a new instance at `company` requires the existence of instances at class `department` and class `supplies` that form the ternary relationship described by the constraint. However, we cannot require the pre-existence of such instances, because similar constraints hold for the other two classes. We cannot overcome this cyclic problem, unless we delay constraint checking until all the participant instances of the relationship are created. This requires an architecture with a form of transaction which delays constraint checks until several updates have been made [10]. Currently this facility is not available in ADAM.

There are similar problems with the strictness of numerical quantifiers and most importantly with the `exists exactly` one. For example, the following constraint:

```
exist exactly 1 s in staff
  such that position of s=''Head of Department''
```

does not allow the Head of Department to change ever, since in order to do so, the slot `position` of the former Head of Department must be updated with another string and the same slot of the new one, must be updated to the string "Head of Department". Again this is caused by a lack of transactions.

7.4. Universal constraints

Another restriction of CoLan is that the variables of the quantification part must range over an entity-class or an attribute of an entity-class. Thus CoLan cannot express general constraints about simple objects or undefined attributes. For example, we cannot state theorems about integers or restrictions for an attribute named `salary`, regardless of the class on which it is defined.

The main reason why general constraints cannot be implemented is fundamental for ADAM. Integers, strings, etc., are not distinct objects in ADAM as they would be in e.g. SmallTalk and slots are not considered as objects but only as value depositories. Thus, general constraints cannot be directly attached to distinct classes and therefore, the method invocation approach of CoLan cannot be used for them. Instead code for general constraints would have to be stored separately as a general purpose procedure. However, this is not allowed in object-oriented systems since it violates the encapsulation of objects. Furthermore, general procedures would have to be searched and executed every time an update is about to happen, because the context of their activation is not determined by the class they belong to. This would cause a frequent waste of computational power.

8. Conclusions

In this paper we have described the CoLan language and its implementation. More details on the technical aspects of CoLan can be found in [1]. CoLan incorporates a constraint description and maintenance system that translates CoLan language expressions into Prolog code. This code is attached to specific methods of specific classes of a schema and is stored in the corresponding slots of the schema for future use. Each constraint is stored as a separate object and its object identifier (constraint identifier) is used to link it to the generated code. When an update is attempted the system is responsible for retrieving the appropriate pieces of code and checking the validity of the update. If the update satisfies every applicable constraint then it is accepted, otherwise an informative error message is printed and the update is rejected.

The CoLan language has been implemented and used to enforce constraints against a schema defined using the OODB ADAM, which is built on top of Prolog. Prolog is a very convenient language for generating and transforming code and it works well when generating triggers from declaratively expressed CoLan constraints. However, CoLan is not dependent on Prolog, and it could be implemented in a high-level language which can treat procedures as data, such as SmallTalk. It is not intended for use with deductive databases. Instead, it is strongly influenced by Daplex, the query language of FDM [26].

CoLan contains a mixture of first order logic and functional programming features and supports the use of quantified variables including universal, existential and numerical quantifiers. These variables range over a finite known universe, such as a set of object identifiers of stored objects, or a subrange of integers. The functional aspects of CoLan include the use of function application on variables that represent instances of the quantified variables. Function composition is also supported both in the quantification part and in the predicate part of an expression. CoLan is quite general as a language and its syntax leaves room for future extensions both to the language and the CoLan system as a whole. The language is expressive and user-friendly, so that it can capture most of the semantic constraints that hold between the data of a schema. Our intended end users are scientists working with object databases, and they find the functional syntax clearer and easier to comprehend than the first-order logic style which is appropriate to deductive databases.

CoLan could also be incorporated in the P/FDM system [12], i.e. the Prolog implementation of the Functional Data Model (FDM) [26], since CoLan is closely related to Daplex, the query language of P/FDM. The CoLan expressions could be permanently stored along with the other metadata, and Prolog code could be activated at transaction-time. We are currently pursuing this [10], using a richer transaction model which overcomes the problems with ADAM described in Section 7.3.

There is plenty of future work to be done to extend CoLan. As we have already discussed, what is missing from CoLan is the facility to express constraints in terms of relationships or arbitrary degree with user-defined semantics. Essentially, CoLan needs a relationship handling background in order to express its method definitions for complex semantic constraints. Such relationships have been studied by Diaz [8, 9] who developed the ABEL system in ADAM. Thus a future project could be the translation of CoLan constraints into ABEL rules. CoLan could then become an alternative form of constraint description to the constraint equations used in the original ABEL system [9].

An alternative in ADAM is to use active rules, so that CoLan can turn from a passive filter for updates into a dynamic transaction system based on semantic constraints. The update would be allowed but the action propagation facility of ABEL active rules would be used to try and find appropriate data and operations on them in order to bring the database back into a consistent state, as in [31]. This facility for a *repair mechanism* is needed for large data intensive applications, with complex interacting constraints.

ABEL combined with a fuller constraint language in the form of CoLan could form an excellent object-oriented core for a new expert systems architecture, since ABEL (and ADAM) can model a lot of the default and automated behaviour that makes a system look 'intelligent' from the user's viewpoint. Active rules [7] would serve as an extension to production rules in this architecture where much of the knowledge is stored not in a 'central' program but inside the appropriate classes. Objects would do most of the work by sending messages to each other, so that the expert system would just co-ordinate the actions of the various objects by triggering them and maintaining the high-level flow of control. Such a system should ensure high level semantic expressiveness and the ease of maintenance and evolution that is so desirable in advanced expert systems.

Acknowledgments

We are indebted to Suzanne Embury for many helpful discussions about constraints, her invaluable help with ADAM and contribution to understanding and criticizing some of the referenced related work. We would also like to thank Oscar Diaz and Norman Paton for useful discussions (on the internet) about active rules and object-oriented databases. We also thank the anonymous referees for their valuable comments and suggestions.

The first author was supported (during his MSc studies in Aberdeen University) by an EEC grant (European Social Fund).

Appendix A: Syntax of CoLan

```
⟨constraint⟩::=⟨quantification⟩ [such that] [⟨predicates⟩]
⟨quantification⟩::=⟨quant_exp⟩ [⟨quantification⟩]
⟨quant_exp⟩::=⟨quant_var⟩ [such that] [⟨predicates⟩|⟨constraint⟩]
⟨quant_var⟩::=⟨quantifier⟩⟨var⟩ in ⟨compound_class⟩
⟨quantifier⟩::=forall | [not] exists | exist ⟨numerical_quantifier⟩
⟨numerical_quantifier⟩::=⟨numerical_expression⟩⟨number⟩
⟨numerical_expression⟩::=⟨numerical_symbol⟩|⟨numerical_words⟩
⟨numerical_symbol⟩::=> | >= | < | =< | =
⟨numerical_words⟩::=only | exactly | more than | no less than |
                          at least | less than | no more than | at most
⟨compound_class⟩::=⟨slot⟩ of ⟨compound_class⟩|⟨class⟩
⟨predicates⟩::=⟨conjunction⟩ [or ⟨predicates⟩]
⟨conjunction⟩::=⟨predicate⟩ [and ⟨conjunction⟩]
⟨predicate⟩::=⟨multi_operand⟩⟨relational_connector⟩⟨multi_operand⟩
⟨multi_operand⟩::=⟨operand⟩|⟨complex_operand⟩
⟨operand⟩::=⟨slot_operand⟩|⟨constant⟩
⟨slot_operand⟩::=[⟨slot⟩ of]⟨var⟩|⟨slot⟩ of ⟨slot_operand⟩
⟨complex_operand⟩::=⟨operand⟩⟨operation⟩⟨operand⟩
⟨operation⟩::=+ | - | * | /
⟨relational_connector⟩::= =| ⟨⟩ | > | < | >= | =< | is in | not in
⟨var⟩::=atom, ⟨slot⟩::=atom, ⟨class⟩::=atom, ⟨number⟩::=integer
⟨constant⟩::=integer | string | plog
```

Appendix B: Example university schema in ADAM notation

```
:- new([person, [
         slot(slot_tuple(cname, global, single, total, string)),
         slot(slot_tuple(sname, global, single, total, string)),
         slot(slot_tuple(age, global, single, optional, integer)),
         key([sname, cname])
   ]])⇒colan_class,
   new([staff, [is_a([person]),
         slot(slot_tuple(phone, global, single, optional, integer)),
         slot(slot_tuple(research_interest, global, single, optional, string)),
         slot(slot_tuple(project, global, set, optional, string)),
        slot(slot_tuple(position, global, single, optional, string))
  ]])⇒colan_class,
  new([ra, [is_a([staff]),
        slot(slot_tuple(project supervisor, global, single, optional, staff))
  ]])⇒colan_class,
  new([lecturer, [is_a([staff]),
        slot(slot_tuple(supervised_student, global, set, optional, postgrad))
  ]])⇒colan_class,
  new([student, [is_a([person]),
        slot(slot_tuple(year, global, single, total, integer)),
        slot(slot_tuple(tutor, global, single, optional, staff))
  ]])⇒colan_class,
  new([undergrad, [ is_a([student]) ]])⇒colan_class,
  new([postgrad, [is_a([student]),
        slot(slot_tuple(subject, global, single, total, string)),
        slot(slot_tuple(supervisor, global, set, optional, lecturer))
  ]])⇒colan_class,
  new([pgra, [is_a(postgrad, ra]),
        slot(slot_tuple(percentra, global, single, optional, integer))
  ]])⇒colan_class.
```

Appendix C: Sample interaction with CoLan system

In the next few pages we have included a script file of a sample interaction with CoLan system. The examples are based on the university schema defined in Appendix B. Our comments on the interaction are written in italics. All the user's inputs are beginning with Prolog's prompt '?-'.

Example 1. Insert a constraint in the schema sending a message to the appropriate class.

```
?- put_constraint(['forall p in person
                        age of p>0 and age of p<120'])⇒person.
yes
```

Create a new instance of class `person`.

```
?- new([_, [cname(['Bruce']), sname(['Dickinson']) ]])⇒person.
yes
```

Try to give an invalid age to the person that has been created!

```
?- get(A)⇒person, put_age([130])⇒A.
Error! Message 'put_age' violates the constraint:
```

```
''forall p in person age of p>0 and age of p<120''
which is defined at class 'person'.
no
```

Try to give an acceptable age.

```
?- get(A)⇒person, put_age([100])⇒A.
A=0#person
```

Try to change the age by introducing an invalid one!

```
?- update_age([100, 150])⇒0#person.
Error! Message 'update_age' violates the constraint:
''forall p in person age of p>0 and age of p<120''
which is defined at class 'person'.
no
```

Example 2.

```
?- put_constraint(['forall l in lecturer
                    not exists s in staff
                     such that phone of l=phone of s'])⇒lecturer.
yes
```

Which constraints are defined at class lecturer? (only one)

```
?- get_constraint(A)⇒lecturer.
A='forall l in lecturer not exists s in staff such that phone of l=phone of s';
no
```

Create a new lecturer.

```
?-new([_,[cname(['Bruce']),sname(['Dickinson']),phone([1111])]])⇒lecturer.
yes
```

Try to create a new member of the staff with the same phone number as the lecturer created before. Notice that the constraint is enforced due to constraint inversion (see Section 6.4). Furthermore, the constraint is inherited from class `staff` *to class* `pgra`.

```
?- new([_, [cname(['Dave']), sname(['Murray']), subject(['guitar']),
          year([2]), phone([1111]) ]])⇒pgra.
Error! Message 'new' violates the constraint:
''forall l in lecturer not exists s in staff such that phone of l=phone of s'' which
is defined at class 'staff'.
```

```
no
```

Example 3. This is a very complex constraint!

```
?- put_consrtraint(['forall l in lecturer
                     not exists s in supervised_student of l
                      such that research_interest of l <>
                       research_interest of tutor of s'])⇒lecturer.
yes
```

Create the appropriate instances to demonstrate how the constraint can be violated.

```
?- new([Tutor, [cname(['Ozzy']), sname(['Osbourne']), phone([1111]),
              research_interest(['singing']) ]])⇒staff,
   new([PGRA, [sname(['Alice']), cname(['Cooper']), age([35]),
              tutor([Tutor]) ]])⇒pgra,
   new([_, [cname(['John']), sname(['Demayo']), research_interest(['bass']),
          supervised_student([PGRA])]])⇒lecturer.
Error! Message 'new' violates the constraint:
''forall l in lecturer not exists s in supervised_student of l such that research_
interest of l <> research_interest of tutor of s''
which is defined at class 'lecturer'.
no
```

Only the first two instances have been created, thus the third violated the constraint and its creation has been rejected.

```
?- get(A)⇒person, display⇒A.
Display of 0#staff            Display of 1#pgra
instance_of: staff            instance_of: pgra
Slot values:                  Slot values:
key: [Osbourne, Ozzy]         key: [Alice, Cooper]
sname: Osbourne               age: 35
cname: Ozzy                   sname: Alice
phone: 1111                   cname: Cooper
research_interest: singing    tutor: 0#staff
.....                         .....
A=0#staff;                    A=1#pgra;
no
```

Example 4.

```
?- put_constraint(['exist at least 2 p in person
                    such that age of p>100'])⇒person.
yes
```

The above constraint imposes a lower limit on the number of instances that satisfy it, so in order to demonstrate it we create two instances and then we try to delete one of them. Note that the system assumes constraint is satisfied before insertion (its not!) and only checks on deletion. This is discussed in Section 6.2.3.

```
?- new([_, [cname(['Rob']), sname(['Halford']), age([105])]])⇒person,
   new([_, [sname(['Ronny']), cname(['Dio']), age([110])]])⇒staff.
yes
?- get_by_cname(['Dio'],A)⇒person, delete⇒A.
Error! Message 'delete' violates the constraint:
''exist at least 2 p in person such that age of p>100''
which is defined at class 'person'.
no
```

If the user tries to delete the age of an instance then the constraint is also violated!

```
?- get_by_cname(['Dio'],A)⇒person, delete_age([_])⇒A.
Error! Message 'delete_age' violates the constraint:
''exist at least 2 p in person such that age of p>100''
which is defined at class 'person'.
no
```

Illegal update because the deletion of the old age violates the constraint and the insertion of the new one does not "restore" it.

```
?- get_by_cname(['Dio'],A)⇒person, update_age([_, 95])⇒A.
Error! Message 'update_age' violates the constraint:
''exist at least 2 p in person such that age of p>100''
which is defined at class 'person'.
no
```

The deletion of the old age violates the constraint, but the new age is consistent with the constraint, so the update is accepted!

```
?- get_by_cname(['Dio'],A)⇒person, update_age([_, 109])⇒A.
A=1#staff
```

Delete the constraint. Code fragments will be deleted from every relevant slot.

```
?- delete_constraint(['exist at least 2 p in person
                                   such that age of p>100'])⇒person.
```

Now we can safely delete the instances.

```
?- get(A)⇒person, delete⇒A.
A=0#person;
A=1#staff;
no
```

Example 5. Insert a constraint to demonstrate the peculiar `exists exactly` *quantifier.*

```
?- put_constraint(['exist exactly 2 p in person
                         such that age of p>100'])⇒person.
yes
```

Re-create the instances! Notice that the system does not complain although constraint is not pre-satisfied (see Section 6.2.3).

```
?- new([_, [cname(['Rob']), sname(['Halford']), age([105])]])⇒person,
   new([_, [sname(['Ronny']), cname(['Dio']), age([110])]])⇒staff.
yes
```

Try to insert a new instance. The new constraint insists that exactly two instances should satisfy it!

```
?- new([_, [cname(['Tony']), sname(['Iommie']), age([110])]])⇒staff.
Error! Message 'new' violates the constraint:
''exist exactly 2 p in person such that age of p>100''
which is defined at class 'person'.
no
```

Now try to delete an instance that satisfies the constraint.

```
?- get_by_cname(['Dio'],A)⇒person, delete⇒A.
Error! Message 'delete' violates the constraint:
''exist exactly 2 p in person such that age of p>100''
which is defined at class 'person'.
no
```

We think that the above examples demonstrate our work adequately.

Appendix D – Example code fragments

We will now present some examples of Constraint Compilation into Prolog. The code illustrated is some times simplified for illustrative purposes. The code that we give is for various methods affected by the corresponding constraint. The methods are named by the Prolog functor starting put_, update_, etc.

Example 1.

```
forall l in lecturer
  age of l > 25 and age of l < 70
```

The above constraint has the very simple translation in Prolog for method put_age:

```
put_age(. . . , [Age]) :-
     (not(Age > 25); not(Age < 70))→
         (error_message(. . .), fail);
         age ←Age.
```

Therefore, the above constraint's Prolog equivalent is translated as:
If *not lecturer's age within range*
then *print an error message* **and** *reject the update*
otherwise *update the age of the recipient object.*

Example 2.
Existentially quantified constraints need more complex code, because we have to count the number of instances that satisfy certain criteria and test it according to the quantifier. For example consider the constraint:

```
exist at most 2 p in person
  such that age of p > 100
```

which has the following translation for method put_age:

```
put_age(. . . , [Age]) :-
    (Age > 100→
        (findall(Inst, (get(Inst)⇒person,
                         get_age(AgeInst)⇒Inst,
                         AgeInst > 100),
                List),
         length(List, Length),
         NewLength is Length + 1,
         (not(NewLength = <2)→
            (error_message(. . . .), fail);
            true));
       true),
   age ←Age.
```

A notable thing is the check for Age > 100 before the global check is performed and the trivial satisfaction of the constraint if this is not so.

Example 3.
Constraints with two variables produce a little more complex code, as they require the two variables to be different, when they range over the same entity-classes or over classes that are 'is-a' related. The following constraint is a typical example:

```
forall r in ra
  exist at most 2 rl in ra
    such that phone of r = phone of rl
```

The translation of the above constraint is:

```
put_phone(. . . , [Phone]):-
    message_recipient(Id),
    findall(Inst, (get(Inst)⇒ra,
                   Inst \== Id,
                   get_phone(PhoneRa)⇒Inst,
                   Phone== PhoneRa),
            List),
    length(List, Length),
    not(Length=< 2)→
        (error_message(. . . .), fail);
        phone←Phone.
```

The interesting bit here is the check Inst\== Id that ensures that the two variables stand for different instances.

Example 4.
This constraint also has two variables, but unlike the above, the second variable depends upon the first one:

```
forall l in lecturer
  not exists r in research_assistant of l
    such that research_interest of l <>
              research_interest of tutor of r
```

The difference between the code generated for this and the previous constraint is inside the findall loop:

```
put_research_interest(. . . , [RI]):-
   message_recipient(Id),
   findall(RA, ( get_research_assistant(RA)⇒Id,
                 get_tutor(Tutor)⇒RA,
                 get_research_interest(RI1)⇒Tutor,
                 RI ⟨⟩ RI1),
             List),
   length(List, Length),
   not(Length < 1)→
      (error_message(. . .), fail);
      research_interest←RI.
```

Example 5.

Finally, this example demonstrates how the values of the slots involved in the predicates are obtained. In section 6, these values have been collectively obtained by the macro *get_values_for_p*$_i$. Consider the following constraint:

```
forall s in student
  age of s > year of s
```

which produces the following code fragments for methods `put_age` and `put_year` of class `student`, respectively:

```
put_age(. . . , [Age]):-
   message_recipient(Id),
   (get_year(Year)⇒Id,
    not(Age > Year))→
       (error_message(. . .), fail);
       age←Age.
```

```
put_year(. . . , [Year]):-
   message_recipient(Id),
   (get_age(Age)⇒Id,
    not(Year < Age))→
       (error_message(. . .), fail);
       year←Year.
```

Notice that the source of the values for the predicate may be the method parameters and/or the instance variables of the message recipient object.

Appendix E – Using meta-classes to support constraint checking

In CoLan, constraint checking is embedded inside the corresponding methods, as we demonstrated in the Section 5. In order to include such a call inside any method that could possibly violate a constraint we had to employ the powerful concept of ADAM meta-classes, that can be used by a programmer to extend ADAM without needing to alter its source code [25].

We introduce two meta-classes: colan_class and colan_meta_class. In fact, colan_class is an instance of the meta-class colan_meta_class (Fig. 2). The introduction of such a high level meta-class (itself an instance of meta_class, the default ADAM meta-class) was necessary because constraints can be inserted at class creation time and not just by using put_constraint. When a user-defined class is created, message new is sent to colan_class, which searches in its meta-class for the appropriate method definition. The method new defined on colan_meta_class extracts the list of constraints defined along with the user-defined class, invokes the default ADAM class-creation method (defined at meta_class), and then uses the already described put_constraint method of colan_class to insert the constraints to the newly created class.

E.1. *Creating a new instance*

Constraints can be violated when new instance objects are created (i.e. when new data is inserted). Therefore, we must ensure constraint checking when message new is sent to a class to create a new instance. The search for the definition for method new for a class begins with the meta-class colan_class, whose instance is the class in question. This definition for new in colan_class first inherits and checks all the constrains, as described, and then invokes the default instance creation method from the mixins class_mixin, etc.

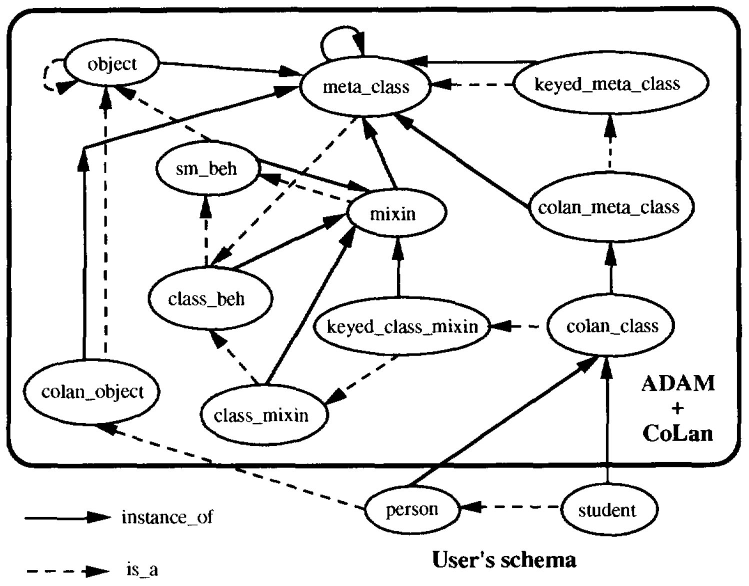


Fig. 2. Pre-defined CoLan & ADAM Meta-classes.

E.2. Deleting an instance

In ADAM instances are deleted by sending them the message `delete`. The search for the method body begins at the class whose instance is the message-recipient object. Unfortunately, the definition of `delete` is not stored in any of the classes of the schema but is inherited from the class `object`. The class `object` is the superclass of all the classes. When a class does not have an explicit superclass, ADAM makes it a subclass of class `object` by default. Therefore, we need a way to override this default `delete` method defined at class `object`. The only way to do this without destroying ADAM's default definitions is to insert a specialisation of class `object` between class `object` and every class of the schema. The purpose of this class, called `colan_object`, is to re-define method `delete` to include a call to `check_constraints` (Fig. 2).

E.3. Updating the internal status of an instance

The rest of methods that can violate a constraint are the ones that affect the internal state of an instance, i.e. methods that put, delete, or update the value of a slot of an instance. Thus the call to the predicate `check_constraints` must be a part of the first occurrence of every method definition, for all the classes and all the slots. The meta-class `colan_class` is responsible for altering every class method that accesses instance variables, and it either includes this call at the beginning of the first occurrence of every class method body when a class is created, or it includes a call to inherit the superclass' method.

Note that the call is included only in the method body of the first occurrence of the method in the class hierarchy. Subsequent subclasses inherit the superclass method definition and therefore the call to `check_constraints`. This is necessary in order to avoid redundant checks of constraints.

But this does not allow for the re-definition of a method by a user, later in the life-cycle of the database. The user might forget to include the call to `check_constraints`, which is the one that guarantees the consistency of the database. In order to overcome this problem, we had to alter the behaviour of those meta-class methods that are responsible for maintaining the usual class methods. These are the methods `put_method`, `replace_method` and `delete_method` whose default definition is stored in a mixin called `sm_beh` [8]. The meta-class `colan_class` is 'is-a' related to the `sm_beh` mixin (Fig. 2), so in order to override the default behaviour of these meta-methods we had to include overriding definitions in the `colan_class` class descriptor. The fact that we were able to do this is a tribute to the extensibility of ADAM through meta-classes.

References

[1] N. Bassiliades, Constraint description in ADAM, M.Sc. Thesis, University of Aberdeen, 1992.

[2] N. Bassiliades and I. Vlahavas, Modelling constraints with exceptions in object-oriented databases, to appear in *13th Int. Conf. on The Entity-Relationship Approach*, Manchester, UK (Dec. 1994).

[3] R.J. Brachman, I lied about the trees or, defaults and definitions in knowledge representation, *AI Mag.* 6 (1985) 60–93.

[4] F. Bry and R. Manthey, Checking consistency of database constraints: A logical basis, *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan (1986).

[5] S. Ceri and J. Widom, Deriving production rules for constraint maintenance, *Proc. 16th Int. Conf. on Very Large Data Bases*, Brisbane, Australia (Morgan-Kaufmann, 1990) 566–577.

[6] A. Chang, Deduce 2: Further investigations of deduction in relational databases, in: H. Gallaire and J. Minker, eds., *Logic and Databases* (Plenum Press, NY, 1978) 201–236.

[7] O.G. Diaz, N.W. Paton and P.M.D. Gray, Rule management in object-oriented databases: A uniform approach, in: F. Saltor, ed., *Proc. 17th Int. Conf. on Very Large Data Bases*, Barcelona, Spain, (1991) 317–326.

[8] O.G. Diaz, From ADAM to ABEL: Making object-oriented databases more knowledgeable, Ph.D. Thesis, University of Aberdeen, 1991.

[9] O.G. Diaz and S.M. Embury, Generating active rules from high-level specifications, in: P.M.D. Gray and R.J. Lucas, eds., *Advanced Database Systems – Proc. 10th British Nat. Conf. on Databases*, Aberdeen, Scotland (Springer-Verlag, 1992) 227–243.

[10] S.M. Embury, P.M.D. Gray and N.D. Bassiliades, Constraint maintenance using generated methods in the P/FDM object-oriented database, in: N.W. Paton and M.H. Williams, eds., *Proc. 1st Int. Workshop on Rules in Database Systems*, Edinburgh, Scotland (Springer-Verlag, 1993) 364–381.

[11] A. Goldberg, D. Robson, *Smalltalk-80, The Language and Its Implementation* (Addison-Wesley, 1983).

[12] P.M.D. Gray, D.S. Moffat and N.W. Paton, A prolog interface to a functional data model database, in: J.W. Schmidt, S. Ceri and M. Missikoff, eds., *Advances in Database Technology – Proc. 1st Conf. on Extending Data Base Technology*, Venice, Italy (Springer-Verlag, 1988) 34–48.

[13] P.M.D. Gray, K.G. Kulkarni and N.W. Paton, *Object-oriented databases, A Semantic Data Model Approach* (Prentice-Hall, 1992).

[14] P.W.P.J. Grefen and P.M.G. Apers, Integrity control in relational database systems — An overview, *Data & Knowledge Engrg.* 10 (1993) 187–223.

[15] P.W.P.J. Grefen, Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem, in: R. Agrawal, S. Baker and D. Bell, eds., *Proc. 19th Int. Conf. on Very Large Data Bases*, Dublin, Ireland (Morgan Kaufmann, 1993) 581–591.

[16] H.V. Jagadish and X. Qian, Integrity maintenance in an object-oriented database, in: L.Y. Yuan, ed., *Proc. 18th Int. Conf. on Very Large Data Bases*, Vancouver, Canada (Morgan Kaufmann, 1992) 469–480.

[17] Z. Jiao and P.M.D. Gray, Optimisation of methods in a navigational query language, in: C. Delobel, M. Kifer and Y. Masunaga, eds., *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, Munich, Germany (Springer-Verlag, 1991) 22–42.

[18] K.G. Kulkarni and M.P. Atkinson, EFDM: Extended functional data model, *Comput. J.* 29 (1986) 38–46.

[19] J.M. Nicolas and K. Yazdanian, Integrity checking in deductive databases, in: H. Gallaire and J. Minker, eds., *Logic and Databases* (Plenum Press, NY, 1978) 325–344.

[20] J.M. Nicolas, Logic for improving integrity checking in relational databases, *Acta Informatica* 18 (1982) 227–253.

[21] G.M. Nijssen, Modelling in data base management systems, in: P.A. Samet, ed., *Proc. Euro IFIP 79*, London (North-Holland, 1979) 39–52.

[22] N.W. Paton, ADAM: An object-oriented database system implemented in Prolog, in: Williams, ed., *Proc. 7th British Nat. Conf. on Databases* (Cambridge University Press, 1989) 147–161.

[23] N.W. Paton and O.G. Diaz, Meta-classes in object-oriented databases, in: W. Kent and R. Meersman, eds., *Object-Oriented Databases: Analysis, Design and Construction* (North-Holland, 1990).

[24] N.W. Paton and O.G. Diaz, Object-oriented databases and frame-based systems: A comparison, *Information and Software Technology* 33 (1991) 357–365.

[25] N.W. Paton, O.G. Diaz and M.L. Barja, Combining active rules and meta-classes for enhanced extensibility in object-oriented systems, *Data & Knowledge Engrg.* 10 (1993) 45–63.

[26] D.W. Shipman, The functional data model and the data language DAPLEX, *ACM TODS* 6 (1981) 140–173.

[27] M.L. Stonebraker et al., Third-generation database system manifesto, in: R.A. Meersman, W. Kent and S. Khosla, eds., *Object-Oriented Databases: Analysis, Design and Construction* (*DS-4*) (North-Holland, 1992) 495–511.

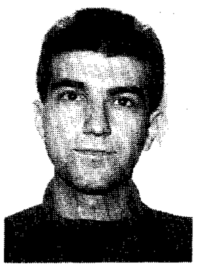
[28] D.A. Turner, The semantic elegance of applicative languages, in *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire (1981) 85–92.

[29] S.D. Urban, Constraint analysis for the design of semantic database update operations, Ph.D. Thesis, University of Southwestern LA, 1987.

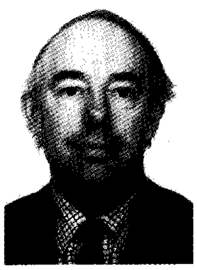
[30] S.D. Urban, ALICE: An assertion language for integrity constraint expression, in: *Proc. Computer Software Applications Conf.*, Orlando (1989).

[31] S.D. Urban, A.P. Karadimce and R.B. Nannapaneni, The implementation and evaluation of integrity maintenance rules in an object-oriented database, in: *Proc. IEEE Int. Conf. on Data Engineering* (1992) 565–572.

[32] A. Van Gelder and R.W. Topor, Safety and translation of relational calculus queries, *ACM Trans. Database Syst.* 16 (1991) 235–278.



Nick Bassiliades received the BSc degree in Physics (1991) from Aristotle University of Thessaloniki (Greece) and the MSc degree in Applied Artificial Intelligence (1992) from University of Aberdeen (Scotland). During his MSc thesis he worked on the definition and enforcement of integrity constraints in the ADAM object-oriented database system. He has also been involved in projects concerning Prolog, databases and parallelism. He is currently pursuing his PhD degree in the Department of Informatics at Aristotle University on the parallel management of knowledge bases, under a scholarship from the Greek State Scholarships Foundation (I.K.Y.). His main interests are deductive object-oriented databases, integrity constraint checking and parallel database systems.
He is a member of the Greek Physics & Computer Societies and a student member of the IEEE/IEEE Computer Society.



Peter Gray is a professor in the Department of Computer Science at Aberdeen University, where he has been since 1968. He originally worked on the ABSYS system where he implemented persistent storage of list-structured objects in a very early equational or constraint programming system. This lead to an interest in logic programming and automatic program generation applied to scientific database systems. Currently he is leading work which applies these ideas to protein structure design and engineering design, using a large object database based on a semantic data model. He has written or co-authored several books on this, including "Logic, Algebra and Databases (1985)".
He received his B.A. in physics from Cambridge (1961) and D. Phil in elementary particle physics and computing from Oxford (1965). He is a fellow of the BCS, and European program chair for VLDB (1995).