Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M.L. Kersten, F.H. Schippers

Towards an object-centered database language

Computer Science/Department of Algorithmics & Architecture Report CS-R8630 September

---

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Towards an Object-Centerd Database Language

Martin L. Kersten, Frans H. Schippers
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

In this report we discuss ongoing research in the area of object-oriented database systems at the CWI. The central theme of this paper is the friction encountered when using an object-oriented (O-O) language, such as Smalltalk, in the database arena. A series of (open) database issues is given for which the object-oriented paradigm does not provide an elegant solution. A refinement of the O-O concepts is given which emphasizes the dynamic classification of objects through its characteristic properties. Our approach is illustrated by a description of the *object-centered* database language *Godel* and its use. A central language concept is the *guardian*, which is a high-level declarative description of a process which algorithmically reacts to states and to state changes of an object base. A prototype implementation of *Godel* has been implemented in C-Prolog.

1980 Mathematics Subject Classification: 69D42, 69H23, 69K14.
1985 CR Categories: D.3.2, H.2.3, I.2.4.
Key Words & Phrases: object-oriented languages, database management, knowledge representation.
Note: To be published in the proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, Sep. 1986.

1. Introduction

The Object-Oriented paradigm represents one of the most successful paradigms in many areas of computer science. It has gained wide acceptance as a unifying paradigm for the design of database systems, programming languages, and artificial intelligence area over the last decade. Notably in this respect is the role of Smalltalk [3] which shows that the concept is of particular importance when dealing with man-machine communication on a high-resolution graphics workstation.
Although the object-oriented paradigm has many different uses and therefore many different definitions, the following aspects are recognized as being essential: [10]

1) *Data abstraction and encapsulation.*
   Every object comes with a set of operations which are used to operate upon and to change the object. Moreover, the object consist of a public interface and a private implementation part.
2) *Object identity*
   The object identity is independent of (mutable) values of properties which makes a representation into a real-world entity. Once it has an object-id it can be referenced by

Report CS-R8630
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

it regardless any change.

3) *Messages*
Contrary to high-level programming languages, objects communicate by passing messages. Each message consists of a receiver object identity, the particular message name and the arguments for the message.

4) *Property inheritance*
Objects are organized into classes, which in turn are organized in class hierarchies. This way economy of specification is achieved by giving a single description of the generic object properties and by automatic inheritance of properties defined at a lower level of abstraction.

5) *Overloading*
Operators (function and procedure) can be overloaded. Both the operator name, the argument types, and the class of the receiver in the type hierarchy determines the specific operator definition.

6) *Late binding.*
Moving the binding of variables to runtime improves the expressiveness at the cost of loosing compile-time error checking capability.

7) *Graphics*
The dialogue with modern graphical workstations require loose control over the input sequence. An object-oriented approach is best suited to model this manipulation of individual objects presented on a screen.

As mentioned before, one of the areas where the object-oriented (O-O) programming paradigm has gained momentum is the database research area. The emphasis placed on the O-O aspects can best be illustrated by some of the ongoing projects in this area.
The first project to mention is the TAXIS in Toronto, where many aspects of the object-oriented programming paradigm has been explored. In particular, it has been used to model office environments and the design of conceptual data models [8].
A more direct approach to apply the O-O paradigm to the database arena has been taken by Maier et al [2]. They try to model the database management system directly after the Smalltalk language and its man-machine style [7]. An increasing stream of activities can be recognized in the area of expert database systems. In particular, the design and implementation of knowledge base management systems. Example projects in this field are the constraint maintenance system PRISM [9] and the object-oriented database management system Sembase [6].

In this paper we review some of the problems which lead to the increased interest in the object-oriented paradigm as exemplified by workable implementations of the language Smalltalk. We will argue that despite its desirable properties an object-oriented approach, as characterized above, leaves many database problems unresolved. As a step towards an adequate solution we suggest to switch focus to the characteristic properties of objects as they emerge during their life span and organize the processing around it. In contrast with the O-O approach this means that the class of an object becomes a dynamic property and that message passing between objects is more elaborated than suggested by the Smalltalk implementation. Such an approach we call *object-centered*, because it is the object representation of a real-world entity which determines its semantic properties rather than the class in which it is created.

The rest of this paper is organized as follows. In section 2 some open database issues are formulated in an object-oriented context. Section 3 presents an overview of the object-centered database language Godel. In section 4 three examples are given to illustrate the effectiveness of our approach in dealing with cooperative processes, evolutionary databases and object classes descriptions.

2. Database problems in an object-oriented context

In this section we review part of the open database problems and interpret them in an object-oriented context. In this discussion we limit ourselves to comparisons with Smalltalk (ST-80). This language can be considered a prototypical example of an object-oriented language. We are aware of variations of Smalltalk which address some of the issues raised in this section, but do not intend to give a full account of such approaches here.

An essential aspect of a database system is the data model it supports. The data model determines the data types, the operators applicable and the integrity constraints to be preserved. Contemporary database systems use the relational data model or an enhancement of it (semantic data model). These three data model dimensions are used to illustrate some open problems and frictions with an object-oriented approach.

Data type

It is generally recognized that the relational model of data is insufficient to easily represent real-world semantics. In particular, the assumption that all objects can be represented as tuples in relations results in complex intra-relationship constraints. For example, CAD/CAM objects exhibit a hierarchical structure which is removed by a mapping to the relational model. The effect is that structure relationships recognized in reality among component has to be regained by extensive use of (recursive) joins.

The prime advantage claimed of an object-oriented approach to database systems is precisely on this topic. Because it provides the framework to define both the structure and the applicable operators of new objects in a consistent manner, hiding implementation details as much as possible. So far no friction with the O-O framework. Problems emerge if we consider the life cycle of an entity in the real-world as represented by an object in an O-O database system.

The first problem to consider is the need for multiple overlapping views of an entity, i.e. opaque datatypes. Two cases can be distinguished. When the view is a re-ordering or partial shielding of the object properties then an object type hierarchy is sufficient. However, when the view represents the result of combining objects to form a new object then we are faced with insurmountable problems, such as view updates, renaming of *methods* to resolve ambiguous property definitions, and union (in)compatibility of composite object classes. For example, consider the two object classes representing departments(depnr, depname) and managers(name, depnr). Then the construction of the subclass representing the objects depmgr(depname, mgrname) requires that a subclass is defined which joins two existing classes. This means that the new class representation should be automatically extended when either one of the super classes instance sets is changed. Moreover, method inheritance becomes ambiguous. In ST-80 such a class can not easily be defined. A solution to resolve ambiguous inheritance paths can be found in LOOPS [1].

The second issue deals with the classification of real-world entities over time. In current database systems an entity is classified once, during its creation. As soon as an entity changes category (=relation type) the old representation is removed and a new object is created. This way the object identity characteristic of an O-O might be violated.
Moving an class instance to a new class, assuming the representation is identical, is not possible in ST-80 either. It is a direct consequence of abstracting the behaviour of objects into hierarchical-structured class definitions and using initialization (and finalization) procedures.

In many cases the object class can be determined automatically from the attribute values of the object or can be explicitly administered as an attribute in the public part of the object representation. As an aside, a tagging of objects with the class names to which they belong provides a uniform solution to many view problems as well. For example, as soon as a person becomes older than 65 it should automatically be classified as a retired person. This requires that the database system contains a *daemon* watching for this event or a

triggering mechanism such that as soon as the age is modified the entity is reclassified. A mechanism to describe this phenomenon within a class definition is conceivable. Yet, ST-80 does not allow it.

Operations.

In ST-80 implementations all objects are addressable, at the language level, by name. In database systems objects are primarily addressed by contents rather than by name. This means that the receiver of a message is a declaratively described collection of objects. Moreover, its effect on message passing is similar to selective broadcasting in operating systems. This means that more control is needed to assemble and manipulate the response set. Clearly, the simplistic message passing protocol in ST-80 is far from such an ideal situation.

The actions on a database take the form of transactions, units of work which preserve the integrity constraints. Moreover, the transaction concept is the unit of recovery from both system and user errors. The transaction concept is not available in Smalltalk. Neither as recoverable unit, nor as mechanism for sharing the object base. Therefore, any database system built upon the ST-80 framework should use the built-in semaphore and process classes and re-implement the traditional database algorithms for transaction management. No direct support is obtained by switching to the a O-O approach.

As mentioned above, a desirable property of database systems nowadays is to be able to define a limited class of triggers. As soon as an update (or retrieval) action is applied to the database a procedure is automatically started to check for inconsistency etc.. Transferred to the ST-80 realm this would mean that one should be able to specify globally which message should be sent as a side-effect of actual, user-activated messages.

The operators are passive in a ST-80 program. They are logically activated by an object receiving a message. Processes are provided as a separated built-in type. The gain from this approach is visibility of control. The disadvantage is that parallelism should be explicitly specified by the programmer. This is a pity because the object-oriented paradigm suggests a way to avoid this by considering each object both as a static and a dynamic (process) entity.

Integrity constraints

Real-world databases are protected against invalid data by integrity constraints. They come in many flavors, such as value range constraints, pattern constraints, ordering and functional constraints. Current database systems only support a limited set. In particular, value range constraints and referential constraints. In ST-80 all constraints have to be described algorithmically by specialized (overloaded) operators. The effect is that constraint maintenance becomes 'visible' through the operator structure. However, the dispersed description of the integrity rules make them less amenable for formal analysis.

One of the interesting aspects in ST-80 is inheritance of properties through type hierarchies. This is convenient for specifying programs, but has some repercussions as well. One of its effects is that a tight link is established among the classes which make extension and modification of the class definition in the future cumbersome. Moreover, it ignores the fact that there are essential two different aspects of inheritance; structural and behavioural inheritance. Structural inheritance is the property that structural details of an object representation should be visible (under constraints) to the outside world, i.e. opaqueness of the representation. For example, when dealing with an aggregated object such as date then structural inheritance allows one to access and manipulate the individual date components, i.e. day, week, year fields.

Contrary, behavioural inheritance makes properties associated with the agents governing the object to become part of the objects property list. For example, the structural properties of a line-printer spooler process are its physical device id, resource requirements and queue length. Each of these can be considered as a behavioural property of the files in the

printing queue. These properties change dynamically as the object is governed by different processes, i.e. line-printer daemon, disk manager or process scheduler.

An object-centered approach

The shortcomings reported above, (and elsewhere), indicate that strict adherence to the object-oriented language features is too restrictive. The prime cause of the friction is the static association of an object with its defining class, regardless the properties it exhibits over time. Moreover, the strict organization of classes into a hierarchy, and the dependencies introduced with it, complicate the manipulation of the semantic descriptions (class definitions). As a solution to the problems we propose an *object-centered* approach, which is characterized by the following features: (using the same framework as for O-O)

1) *Data abstraction and encapsulation.*
   Every object comes with a set of characteristic properties which determine the set of operations used to operate upon and to change the object. Moreover, the object can hide its properties so as to support a private implementation of the applicable operations.

3) *Messages*
   Message passing is asynchronous. The receiver of the message is not (necessarily) known to the sender. The sender places several messages in the database upon which processes (objects) may react. The result of a message can only be determined by observing a particular database state.

4) *Property inheritance*
   The dichotomy of objects (static and dynamic) leads to supporting two kinds of inheritance as well; *structural inheritance* and *behavioural inheritance.* Structural inheritance makes properties of objects structurally associated with an object accessible. Dynamic inheritance specifies the information obtainable by processes, which are determined dynamically, controlling a set of objects.

5) *Overloading*
   Operator definitions (methods) can be overloaded. Both the operator name, the argument types, and the state of the database determine the specific operator definition.

The object-oriented aspects 2), 6), and 7) introduced before remain valid. In the following section we illustrate how these aspects are reflected in the design of the programming language *Godel.*

3. A general object-centered database language

In this section we give a short description of the language *Godel.* The language has been designed to support the construction of knowledge based applications using an object-centered programming paradigm. A detailed description is given in [5]. The language capitalize on experience in earlier object-oriented languages and addresses the database problems posed in the previous section. The prominent *Godel* design considerations are:

1. *Data base management*
   The language deals with a database of objects; integrity of the database is guaranteed through a generalized trigger mechanism; object selection is declarative, based on first-order logic; transaction processing primitives are included.

2. *Knowledge base management*
   Both simple facts (objects) and rules to derive facts are stored in the database in a uniform way. Facts can play roles in different interpretation domains concurrently. Both structural inheritance, i.e. using the value of an objects' subcomponent, and

behavioural inheritance, i.e. using a value obtained from an objects' agent, are provided.

3. *Data flow driven computation*
   Operations are triggered when the operational constraints associated with processes become true in a particular state of the database or when an event is recognized. Primitive trigger conditions are the insertion and modification of objects in the database.
4. *Modularity*
   Facts and rules can be logically grouped into knowledge bases through tagging. The level of integration of multiple knowledge bases accessed during a single session can be precisely controlled. The visibility rules for object properties support information hiding.

The *Godel* design considerations influenced the choices of the particular language features, which are summarized below:

1. *Data paradigm*
   The structure of the data in a database is not static; rather it evolves over time to meet the changing information requirements of users. A declarative specification of the behavioural properties of data forms a proper basis to cope with database evolution. It allows both data and meta-data to be treated in a uniform fashion.
2. *Object-centered paradigm*
   The object-centered paradigm takes the object-oriented approach one step further by making class membership a dynamic property of objects. This way evolution of the knowledge base is accommodated without loosing the ability to safeguard its integrity.
3. *Rule-oriented paradigm*
   The rule-oriented paradigm allows for the description and use of procedural knowledge without specifying in advance all allowable control paths.
4. *Polymorphic typing*
   The language is basically typeless. Variables need not be declared before use. In essence, types are considered 'first class values' and thus may be the result of expressions. A typing system is nothing more than a restricted symbolic evaluation of the program to reveal processing inconsistencies. A dynamic typing system is easily included using the language primitives provided.
5. *Cooperative problem solving*
   Techniques for cooperative problem solving provide the means for distributed knowledge manipulation and forms a basis for contemplating parallel implementation architectures. The integrity of the database is guaranteed by the atomic behaviour of guardians which is implemented through data sharing.
6. *Self-referential*
   An essential aspect of knowledge base systems is their ability to explain their behaviour within the same formalism. The computational model and most language features are documented (and implemented) in terms of itself.

A summary of *Godel.*

The language contains three major building blocks: objects, guardians and functions. The objects are used to model the static aspects of entities. The guardians model processes and entity-classes. The functions define parameterized computations and provide refinements in processes description.

Objects represent entities in the real world, which are stored in a part of the *Godel* system called the object-base, and which are defined by an object definition-statement. We refer to the appendix for a full description of the syntax.

Each object is (individually) described by a set of attributes. An attribute takes either a set of values (the singleton set is allowed) which denote primitive objects or object references. Alternatively an attribute value is assigned a (symbolic) description of how to compute the value, i.e. a deferred parameterless function. Objects have a symbolic name represented by a predefined attribute, which permits manipulation at runtime. New properties of an object can be described at any convenient time by inclusion of a new attribute. Similar, properties can be dropped by removing the attributes. An example of an 'hummingbird' object definition is given below.

examples

```
object 'Artis-Amsterdam' [
    location := 'Plantage Parklaan'
    price := 10
    price.unit := 'Dfl'
]

object hummingbird [
                /* A new object called hummingbird. */
    isa_bird
                /* An attribute without value. */
    food := 'honey'
                /* An attribute with string value. */
    location := 'Artis-Amsterdam'
]
```

The second object has three attributes: isa-bird, food, and the predefined attribute *name*. The value set of the name and food attributes are singleton sets. The isa-bird attribute value set is empty. Note that the object can be recognized as a bird by the occurrence isa-bird tag. The result of the object definition is the extension of the object-base with a new element; the object is added to the knowledge base.

The structural inheritance operators.

The dot-operator ('.') provides access to components of an object. It is similar to the traditional record field denotation in other high-level programming languages. In the first statement of the example below, the value of the food consumed by the hummingbird is retrieved. In this case the value is a singleton set, i.e { 'honey' }. The field denotation can be applied recursively, accessing more detailed descriptions of the object. If the exact path is unknown or all attributes with a given name are required then the dot-dot ('..') operator can be used. Note that the dot and dot-dot mechanism is a form of structural inheritance, details are accessible to the outer levels through name navigation.

examples

```
hummingbird.food              /* denotes { 'honey' }. */
hummingbird.location          /* An zoo object. */
hummingbird.location.name     /* Artis-Amsterdam. */
hummingbird..name             /* Artis-Amsterdam. */
hummingbird..unit             /* 'Dfl'. */
hummingbird.location.location /* Plantage Parklaan. */
```

The modify operators.

Manipulation of attribute (and value) sets is supported by the operators associate (:+) and de-associate(:−). They take the result of an expression and extend (or reduce) the attribute set of the object referenced. Thus, addition and deletion of attributes and values are conceptually identical.

examples

```
hummingbird.food :+ 'water'
                    /* extend the value set. */
hummingbird :+ location
                    /* The attribute location is added. */
hummingbird :− food
                    /* The food-attribute is deleted. */
```

The guardians.

Guardians are high-level declaratively described processes which react on observed states and state changes in the object base. They are similar to daemons, but differ in the way they are introduced and manipulated. A guardian has three distinctive components. First, like objects, it has a name and possibly a set of attributes. Second, it contains a list of rules. Each of which consists of a predicate (with free variables) and a statement block. The predicate is evaluated continuously against the current object base. As soon as a binding of the variables is found for which the predicate is true the statement block is executed. To simplify the specification (and the optimization) of the guardians the common part of the rule constraints can be factored out and placed at the beginning; in the form of a scope expression. Third, the guardian terminates with an optional access constraint which regulates access to this guardian viewed as a static object.

The statement blocks associated with guardian rules are interpreted as transactions. They move the object base from consistent state to consistent state. An undo operator is provided to the programmer to express recovery of user actions. The undo is automatically called when the statement block can not be executed with success. It is assumed that concurrent transactions are coordinated by a transaction manager using traditional database techniques.

The guardian shown below watches the database for birds. It ensures that the food attribute of birds is set. The global constraint limits the search space to objects with the isa-bird attribute. The detailed rule reacts to a particular unpleasant situation, i.e the food attribute set is empty. The guardian has an attribute which is incremented each time a new bird object is added to the object base. This attribute can be considered a behaviour property of bird objects. Because the value is relevant for all birds and maintained by the agent looking after the birds objects. This attribute value is accessible through the hat ('^') and hat-hat ('^^') operators.

examples

```
guardian birdwatcher
when O.isa_bird [       /* Watches for birds */
    birdcount := 0      /* counter for all birds */
    when O.food = {} [
        /* incomplete information has be localized */

        write ('What it the food of ', O.name, '\n')
        read O.food
    ]

    when :+(O) [  /* triggered by successful insertion */
        birdcount := birdcount + 1
    ]
]
```

Variables and expressions.

The lexical conventions used is inspired by Prolog. Objects (attributes, guardians and functions) are named by identifiers (or string constants). Variables are recognized as identifiers starting with an upper case character. The variables are used in the same way as object names; they give access to objects. The lexical scope of the variable is limited to the lexical scope determined by the closest enclosing brackets. In use there is no difference between names and variables.

Variables have two states; bounded and unbounded. The first occurrence of a variable defines it and turns it in an unbounded variable. Once a variable is assigned a reference to an object through an expression is becomes bounded. Automatic variable binding is provided as a side-effect of expression evaluation. When a variable is introduced in an expression then it is bound with an object in the object-base such that the factor (in which it occurs) does not denote the null object. This way a shorthand is obtained for specifying an existentially quantified expression. For example, in the statements below the variable Bird is bound to *some* bird in the database. It is unknown which particular bird. Selective binding of variables is provided through the unique and set construct. The set constructor can also be used to declaratively select object sets for modification, i.e. occur at the left hand side of an assignment statement.

In the scope-expression all possible variable bindings are explored to satisfy the constraint. The variables for which the predicate is true remain bounded till the end of the associated statement block.

examples

```
Bird.isabird
        /* binds with some bird. */
( Bird : Bird.isabird and Bird.food = 'honey' )
        /* binds with exactly one honey eating bird. */
{ Bird : Bird.isabird and Bird.food = 'honey' }
        /* binds with all honey eating birds. */
```

Statements.

To describe algorithmic actions in a the *Godel* program some well-known control structures are introduced. A statement block is provided which enforces a sequential execution of the statements listed. The elements of the statement block are assignments, if-statements, repeatitive statements, input/output statements and function calls. The assignment statement have been described before and need no further introduction.

The if-statement is a sequential interpreted list of qualified statement blocks. In the if-statement at most one statement-block is executed, that statement must have a scope-expression that evaluated to true. The do-statement is a repeated version of the if-statement, after a statement-block is selected, a new block is selected, until no block can be found with a provable scope-expression.

Functions are used to specify computations. Function can be overloaded, the selection of which is based on the scope expression belonging to the function definition. So function selection is based on instances of its arguments and the state of the object-base. It is therefore easy to define all kinds of selection mechanisms such as type-based selection and value-based selection.

examples

```
function isa(Object, Guardian)
when guardian.Guardian <> undef [
    return Guardian.scope(Object)
]
```

4. Example *Godel* programs

In this section we present three example *Godel* program to illustrates the language features and usage. The examples illustrate cooperative problem solving, evolutionary database management, and object-class descriptions respectively.

The card game

This section presents an informal and intuitive introduction of the features of *Godel* by means of 'a card game'. Seated at the table are three players, called Shorty, Fat Boy, and Sue. A third person, the arbiter, shuffles a deck of cards and arrange them in a rectangle of 4 rows and 13 columns. When this is finished the arbiter takes three pieces of paper and writes down a task for person.
The task for Shorty becomes:

> *For every two different cards in the same column exchange them such that the clubs occupy the first row, diamonds the second, hearts the third, and spades the fourth row.*

The task for Fat Boy and Sue becomes:

> *For every card not in the proper column select a card from the proper column (and which is not at the correct column either) and exchange them.*

As soon as the players receive their (private) goal the system is set into motion. All players move until no more actions are possible. The player who moves least recently is the winner.

This trivial game, of course, sorts all the cards by suites and value. The points of interest are that all players (processes) can work in parallel without knowledge of each others' task; the actual sequence of actions is determined by the state of the table (=database); the tasks are described by a high level declarative rule selecting the interested database states and a simple algorithm action; concurrent access is synchronized through an exclusive locking scheme. Below a sketch is given of the *Godel* description for this game.

examples

```
object card[ x:=1 y:=1 color:=clubs number:=4 ]
    /* 50 more cards */
object card[ x:=13 y:=4 color:=spades number:=1 /*ace*/ ]

function exchange(O, P)
when O <> undef and P <> undef and P <> O [
    V := O; O := P; P := V
]

guardian 'Shorty'
when C1.x = C2.x and C1 <> C2 and
        C2.color <> C1.color [
    when C1.color = clubs and C1.y <> 1 [
        exchange(C1.y, C2.y)
    ]
    when C1.color = diamonds and C1.y <> 2 [
        exchange(C1.y, C2.y)
    ]
    /* the other two cases are similar. */
]

guardian 'Sue'
when R1.x <> R1.number [
    when R2.x = R1.number and
            R2.number <> R1.number [
        exchange(R1.x, R2.x)
        exchange(R1.y, R2.y)
] ]

guardian 'Fat Boy' := 'Sue'
```

A small relational database

To illustrate the power of *Godel* some pieces of a relational database application are implemented below. The structure of the relational model is mapped to objects tagged with the name of the relation they belong to. As the *Godel* user is free to construct such an object, it should be ensured that each object has the proper attributes. In the guardian *employee_class*, described below, omitted attributes are automatically introduced. Garbage is not removed, but told about. A similar guardian can be defined for retired persons.

examples

```
guardian employee_class [
when O.employee [
    Garbage := { O.A } − { employee, age, pname,
            address, manager, birthyear }
    do Garbage.E [
        write (E.name,
                ' is an illegal employee attribute \ n')
    ]
    Missing := { employee, age, pname, address,
            manager, birthyear } − { O.A }
    do Missing.A [
        O :+ A /* extend attribute set. */
    ]
] ]

guardian retired_class [
when O : O.retired [
    Garbage := { O.A } − { employee, age, pname,
            address, birthyear }
    do Garbage.E [
        write (E.name,
                ' is an illegal retirement attribute \ n')
    ]
    Missing := { employee, age, pname,
            address, manager, birthyear } − { O.A }
    do Missing.A [
        O :+ A /* extend attribute set. */
    ]
] ]
```

Two integrity constraints are implied by the relational model. Namely, each tuple in a relation should be unique and each attribute value is atomic. These constraints can be checked by a single guardian. To make it work we need some information to single out the tuples from other *Godel* object structures. Therefore, assume that all relation names are assembled in a single object.

examples

```
object relations [ employee, retired ]

guardian relational_constraints
when T.Relname = relations.Relname [

    when count( {A.value : T.A} ) <> 1 [
        write(T.name, T.A.name,
                ' violates atomicity property \ n')
    ]

    when T2.Relname <> T.Relname and
            equaltuple(T2, T1) [
        write ('tuple ', T.name, '[')
        do T2.A write(T2.A, ' ')
        write('] violates tuple uniqueness property \ n')
    ]
]
```

Some application specific constraints can be implemented directly using operator overloading or by using a guardian to watch for undesirable situations. The latter approach is illustrated below which warns the user whenever the value of the birthyear becomes invalid.

examples

```
function showtuple(T)
when relations.Relname = t.Relname [
/* show the tuple. */
]

guardian semantic_constraints
when T.Relname = employee or T.Relname = retired [
    when T.birthyear < 1900 or T.birthyear >= 2000 [
        T.birthyear := undef
        showtuple(T)
        write(' violates birthyear constraint \ n')
    ]

]
```

An object class

To illustrate how a Smalltalk-like class definition can be mapped to *Godel* we consider the concept of a Point and Rectangles. A Point represents an x-y pair of numbers usually designating a pixel in a Form. Points refer to pixel locations relative to the top left corner of the Form. By convention, x increases to the right and y downwards. In Smalltalk the class Point is predefined and provides facilities to create points, accessing the individual coordinates, point comparison, point arithmetic, and point functions (distance, transpose, etc.). The class Rectangle represent rectangular areas of pixels. Arithmetic operations take points as arguments and carry out scaling and translation to create new rectangles. Thus, the Rectangle class inherits the Points properties for manipulation of the rectangle corner and center points.

The mapping to *Godel* objects is straightforward. First we have to design an object structure to represent points and rectangles. Below a prototypical Point and Rectangle are presented. Note that this definition does not enforce rectangles to be initialized with points. The functionality is obtained by defining a set of overloaded functions.

Unlike ST-80 the resulting class definitions are flat. That is, all overloaded functions are defined at the same level of abstraction. However, they are coordinated by different scope expressions, which particularize the instance to be taken in each case. The advantage of this approach is that new operator definitions can readily be included. Provided the scope expression denote complementary subsets. In case overloaded functions share objects, there exist multiple applicable function definitions, the user has to extend the built-in function which selects the proper function.

examples

```
object point
[
    xcor := 0
    ycor := 0
]

function newpoint(Xcor, Ycor)
when Xcor.name = 'integer' and Ycor.name = 'integer' [
    return object point [
        xcor := Xcor
        ycor := Ycor
    ]
]


function <(Point1, Point2)
when Point1.name = 'point' and Point2.name = 'point' [
    return Point1.Xcor < Point2.Xcor and
           Point1.Ycor < Point2.Ycor
]
```

5. Summary and future research

In this paper we have presented an overview of the object-centered programming language *Godel*. The rationale for its development has been partly motivated by presenting some current database problems. It shows that a more object-centered description of a data base results in greater flexibility of modelling the real world. In contrast with the O-O approach this means that the class of an object becomes a dynamic property and that message passing between objects is more elaborated then suggested by the Smalltalk implementation. Besides modelling flexibility the language features simplify the definition of evolutionary database systems [4].

A functional prototype *Godel* processor has been implemented under BSD 4.3 in C-Prolog. Our initial findings working with this implementation proves the validity of the language features. However, the performance of the examples shown before needs more work. In particular, a more thorough symbolic analysis of the program is required to reduce the excessive overhead in finding qualifying objects for guardians. And parallel processor architectures are considered for improved execution speed.

The second area of attention is language functionality and the programming methods to be used. Programming in the *Godel* is complex because it is not always clear in advance what the combined, non-deterministic effect is of running the guardians in parallel. Wherever determinism is required it should be encoded in the application. A particular area where these aspects are currently being studied is the design of inference engines in current language framework. Initial investigations provide suggestive evidence that our language is particularly suited for the description of different kinds of guardian supervisors, i.e. inference engines.

References

[1] Bobrow, D.G. and Stefik, M., "The LOOPS Manual," *Xerox Corporation*, 1983.

[2] Copeland, G. and Maier, D., "Making Smalltalk a Database System," *ACM SIGMOD*, pp.316-325, 1984.

[3] Goldberg, A. and Robson, D., *Smalltalk-80 The language and its implementation.* Addison-Wesley, 1983.

[4] Kersten, M.L. and Schippers, F.H., "Using the Guardian Programming Paradigm to Support Database Evolution," *Submitted for publication*, May 1986.

[5] Kersten, M.L. and Schippers, F.H., "*Godel*, A General Object-Centered Database Language", CS-R8615, Centre for Mathematics and Computer Science, April 1986.

[6] King, R., "A Database Management System Based on the Object-Oriented Model," *Proc. Int. Workshop on Expert Database Systems*, pp.443-468, Oct. 1984.

[7] Maier, D., Nordquist, P., and Grossman, M., "Displaying Database Objects," *First Int. Conf. on Expert Database Systems*, pp.15-30, April 1986.

[8] Nixon, B., "TAXIS 84': Selected Papers", CSRG-160, june 1984.

[9] Shepherd, A. and Kerschberg, L., "Constraint Management in Expert Database Systems," *Proc. Int. Workshop on Expert Database Systems*, pp.309-332, Oct. 1984.

[10] Zaniolo, C., Ait-Kaci, H., Beech, D., Cammarata, S., Kerschberg, L., and Maier, D., "Object-oriented Database Systems and Knowledge Systems," *Workshop on Expert Database Systems*, pp.50-65, 1984.

syntax

brackets ::= '[' | ']' | '(' | ')' | '{' | '}' .

operator ::= '<' | '>' | '<=' | '>=' | '<>' | '=' | '?' | '+' | '-' | '*' | '/'
| ' := ' | ' :+ ' | ' :- ' | '.' | '..' | '^' | '^^'.

wordsymbols ::= **access** | **and** | **div** | **do** | **function** | **guardian** | **if**
| **mod** | **not** | **object** | **or** | **read** | **return** | **undo** | **when** | **write.**

comma ::= ',' | /* empty */.

semicolon ::= ';' | /* empty */.

string ::= "" { letter | digit | escaped } "" .

escaped ::= '\n' | '\f' | '\' digit digit digit .

number ::= integer | float .

object ::= **object** (objectname) attributelist (accessrule)
| **object** objectname assignment (accessrule)
| **object** objectname (accessrule) .

attributelist ::= '[' { attributedef semicolon } ']'.

attributeref ::= objectname { '.' attributeref } .

attributedef ::= attributeref | attributeref assignment .

objectname ::= identifier | string.

function ::= **function** fcnheader (scope) (body) .

fcnheader ::= fcnname '(' { parameters } ')' .

fcnname ::= identifier | wordsymbols | operator .

parameters ::= variable { comma variable }.

scope ::= **when** qualifiedexpr.

body ::= '[' { functionstmt semicolon } ']'.

functionstmt ::= statement .

guardian ::= **guardian** (objectname) guardianbody (accessrule).
| **guardian** objectname assignment (accessrule).

guardianbody ::= (scope) rules .

rules ::= '[' { rule semicolon } ']' .

rule ::= attributedef
| **when** qualifiedexpr guardianblock .

guardianblock ::= '[' { guardiansmt semicolon } ']' .

guardianstmt ::= statement .

attribute ::= attrterm { behaviourprop attribute } .

attrterm ::= attrfactor { structureprop attrterm } .

attrfactor ::= objectname | variable | uniqbinding | setconstructor.

structureprop ::= '.' | '..' .

behaviourprop ::= '^' | '^^' .

accessrule ::= **access** qualifiedexpr .

expressionlist ::= expression {comma expressionlist } .

expression ::= conjunction (**or** expression).

conjunction ::= negation (**and** conjunction).

negation ::= (**not**) comparison | (**not**) exists .

exists ::= attribute | attribute <> *undef* | attribute = *undef.*

comparison ::= sum (compop sum) .

compop ::= '<' | '>' | '<=' | '>=' | '<>' | '=' | '?' .

sum ::= (sign) term (addop sum).

sign ::= '+' | '-'.

addop ::= '+' | '-' | '++'.

term ::= factor (mulop term).

mulop ::= '*' | '/' | **mod** | **div**.

factor ::= constant | attribute | uniqbinding | setconstructor | functioncall.

functioncall ::= objectname '(' expressionlist)'.

constant ::= number | string .

qualifiedexpr ::= expression .

uniqbinding ::= '(' expression (':' expression) ')' .

setconstructor ::= '{' (expressionlist) (':' expression) '}'.

statement ::= object | function | guardian | modifystmt | conditionalstmt
| generatorstmt | functionseq | waitstmt | inputstmt | outputstmt
| expression | **undo** | **return** expression .

block ::= statement | stmtblock .

stmtblock ::= '[' { block semicolon } ']' .

modifystmt ::= attribute assignment .

constructor ::= expression | object | function | guardian | stmtblock .

assignment ::= modop constructor .

modop ::= ':=' | ':+' | ':-'.

conditionalstmt::= **if** qualifiedblock
| **if** '[' qualifiedblock { qualifiedblock} ']' .

qualifiedblock ::= (qualifiedexpr) block.

generatorstmt ::= **do** qualifiedblock
| **do** '[' qualifiedblock { qualifiedblock } ']' .

functionseq ::= objectname '[' { constructor comma } ']' .

waitstmt ::= **wait** qualifiedexpr.

inputstmt ::= **read** inputlist .

outputstmt ::= **write** outputlist.

inputlist ::= attribute | '[' { attribute comma } ']'.

outputlist ::= expression | '[' expressionlist ']' .

godelprogram ::= { block semicolon } .