Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M.L. Kersten, F.H. Schippers

Using the guardian programming paradigm to support database evolution

Computer Science/Department of Algorithmics & Architecture Report CS-R8631 September

Using the Guardian Programming Paradigm to Support Database Evolution

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

A guardian is a high-level declarative description of a process which algorithmically reacts to an observed state or state change of a database. In this paper the role of guardians in the object-centered programming language *Godel* is illustrated. This language has been designed to support the construction of knowledge based applications and adaptive information systems. It is shown that traditional problems such as maintaining the integrity of a database is easily resolved using guardians without constraining the evolutionary paths of the database definition. This is accomplished by relaxation of the class membership rule found in object-oriented languages. In *Godel* class membership is a dynamic property determined by the provable characteristics of the object, rather then the operator applied to create the object. Dynamic class membership separates the classification of the object from its time-dependent semantic behavior. Therefore, integrity of the database can be preserved by a set of guardians which represent the integrity rules and react in an algorithmic fashion on any violation.

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 IFIP TC-2 Working Conference on Knowledge and Data (DS-2), Albufeira, Portugal, Nov. 1986.

1. Introduction

The last two decades have shown many advances in the area of information systems theory and practice. In the theory arena, database modelling has been firmed based in predicate logic; as exemplified by the relational data model. Much research has been centered around the relational data model to capture more real-world phenomena[3, 6] such as various forms of integrity constraints[2] and the reasoning model to be used with it.[5, 1] Moreover, the practitioners have demonstrated the feasibility of the relational model by providing efficient and commercial viable solutions, which are currently penetrating the market at a rapid pace (Oracle, Ingres, DB/2).

Despite its commercial success, it has been recognized for some time that the relational data model is just a step in the proper direction. Namely, the model is biased by a simplistic view of the real-world. Its weaknesses show when complex information systems with many user views should be accommodated and when new application domains are brought under its umbrella. Such as CAD/CAM and office automation. As a result, a plethora of extensions to the relational data model have been published since its inception.[9] Conceptual data models have been developed to bridge the semantic gap left and to hide the intricacies of the relational theory.[4] In this process even the fundamental assumptions underlying the relational model, such as tuple uniqueness and atomic

Report CS-R8631
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

attribute values, have been reconsidered[11, 12]

Unfortunately, the formalization of the data model and the enhancements to the relational model have brought about limited new insights in dealing with the evolution of a database system (in a changing organization). This shortcoming stems both from the limitations imposed on the data model by the underlying theory as from the inability to come up with efficient solutions for enhancing the database management system with a time concept.

To illustrate, we can recognize at least four major areas where database systems and the (relational) data model they support (both at the end-user, conceptual, and internal level) fall short in providing the adequate tools and philosophy.

1 *Theory* The theory developed for the description and analysis of databases requires either a schooled mathematician or an expert system tool to use the information system to its full potentials. Both are scarce resources, hard to find and deal with.

2 *Exceptions* The theory developed for integrity preservation is hard to apply in reality. In addition to the previous point, the theory is limited in scope, i.e. (multi-) functional dependencies, range and occurs checks, and sometimes a rudimentary missing value semantics, is all what is provided. The major drawback, however, is that integrity rules tend to change rather rapidly over time (consider the governmental laws on taxation) and marked with many exceptions. To our knowledge no adequate concise theory has been developed and never will, because part of the problem lies in the restrictiveness of the formalization itself.

3 *Completeness* One of the basic assumptions underlying the relational model (and derivatives) is that the database is structural and semantically complete. The former meaning that no entity can exist in the database which has approximately the necessary attributes (Employees without a salary field are syntactically different from employees with a (yet) unknown salary). Semantic completeness meaning that all entities can be classified as belonging to precisely one particular class. (Prospects aren't clients yet, they may turn out to be business spies). Thereby ignoring the fact that an entity may play multiple, possibly conflicting and contradicting roles in real-life as well. (Just consider the roles of a peer reviewer of a proposal for obtaining a scientific grant).

4 *Adaptability* As soon as a database system becomes operational, it starts exhibiting aging characteristics, leading to a series of patches until it is proclaimed dead. And rebuild with new man and material. Again, the lack of adaptability partly lies in the data model itself. For it does not include the means to support the evolution of the database. Moreover, despite the experience built over the years in designing and implementing these systems, a gerontology of information systems has not been attempted.

In our opinion these problems can only be attacked by a new way of thinking; a new way of looking at how people deal with information in reality. The prominent technique applied there is to assemble bits-and-pieces of information rather then living by the law and spirit of a predesigned information model. This way people not only deal with their limited learning capability and memory recall mechanism, but are also able to adapt to chancing situations quickly.

Without saying, any new approach to the database problems should be firmly based on a variety of formal models without using them as dogmas and thus being bound by their inherent limitations.

In the *Godel* project we envision a database as a large, amorphous collection representations of real-world entities. Some representations exhibit structural relationships and have similar properties, but this is not enforced beforehand. In particular, we envision entities to be represented as objects with a set of value bearing attributes. Moreover, the properties determine its semantics rather then the way they have been created. This view is called an *object-centered* description.[7]

The dynamics are modelled by a data manipulation language which provides associative access to the entity representations and which provides capabilities to modify the database through declarative statements. Procedural abstraction is provided through functions. These are parameterized algorithms and their use is controlled by a predicate on the actual arguments and database state.

Associated with a database is a set of guidelines which describes the semantics of the real-world represented by the database in more detail. These guidelines are less constraining then integrity constraints in a traditional approach. It is possible under a set of guidelines to model an ambiguous database (an employee having two salaries) and it is possible that the database is incomplete (An immigrant upon entering the US does not have a social security number). In our view, non-obedience of the guidelines does not imply that these database states are wrong and therefore should be rejected. Contrary, one should be aware of the situation and deal with appropriately at some point in time.

Each guideline is implemented by a *guardian*, a high-level declarative description of a process which algorithmically reacts to an observed state or state change of a database. That is, whenever an unwanted (or interesting) situation is detected a procedure is triggered to notify a user to ask for clarification, or automatically adjust the objects in the database to obey the law. Guardians can thus been seen as warders of the database.

In this paper we show that the guardian programming paradigm is well-suited to implement the traditional rules of integrity. Moreover, it provides the flexibility to adapt the database and the guidelines to chancing needs. Automatic modification of the database is obtained as a side-effect for free (i.e. yearly salary raises are handled automatically).

The rest of this paper is organized as follows. In section 2 we present an overview of the programming language *Godel* which has been developed to support adaptable information systems and to support knowledge based applications. In section 3 a series of traditional integrity problems is reviewed and solved with the guardian programming paradigm. Finally, in a moment of reflection, we summarize our findings and indicate the roads for further development.

2. 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 .[8] 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' controllers, are provided.
3. *Data flow driven computation*
   Operations are triggered when the operational constraints associated with the 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 by the user 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 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.

3. *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.
4. *Rule-oriented paradigm*
   The rule-oriented paradigm allows for the description and use of procedural knowledge without specifying in advance all allowable control paths.
5 *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.
6. *Cooperative problem solving*
   Techniques for cooperative problem solving provides 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.
7. *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, and 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. Each individual object is fully described by a set of attributes. An attribute takes either a set of values or a (symbolic) description of how the value of the attribute is derived. Most objects have a symbolic name for programming convenience, which is represented by a predefined attribute and thereby permits manipulation at runtime.
New properties of an object can be described at any convenient time by inclusion of a new attribute and value derivation function. Similar, properties can be dropped by removing the attributes. The attribute values are either references to existing objects or primitive objects (string and integer). Objects are defined by the object definition-statement. We refer to the appendix for a full description of the syntax. An example of an 'hummingbird' object definition is given below.

examples

```
object hummingbird [           /* A new object called hummingbird */
   isa-bird                    /* An attribute without value. */
   food := 'honey'             /* An attribute with string value. */
]
```

This 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 undefined. Notice that the isabird attribute is used as a type tag. It is a value at a different level of abstraction. 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 modify operators.

Addition and manipulation of attribute (and value) sets is support with the operators associate (:+) and de-associated(:-). They take the result of an expression and extend (or reduce) the object reference at the left hand side of the operator. Moreover, addition and deletion of attributes and values are conceptually identical. Thereby, accommodating the construction of complex object relationships.

examples

```
object.set :+ { 1, 2 }              /* set equals { 1, 2 }. */
object.set :+ { 2, 3 }              /* set equals { 1, 2, 2, 3 }. */
object.set :- { 2, 4 }              /* set equals { 1, 3 }. */
hummingbird :+ location  /* The attribute location is added. */
hummingbird :- food                 /* The food-attribute is deleted. */
```

The structural inheritance operators.

The dot-operator ('.') is used to access components of an object. It is similar to the traditional field denotation of records found in other high-level programming languages. In the first 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. These operators implement structural inheritance, i.e. structural details are accessible by component name navigation.

examples

```
hummingbird.food            /* denotes { 'honey' }. */
hummingbird.location                /* An zoo object. */
hummingbird.location.type  /* 'zoo' is the type. */
hummingbird.location.name /* Artis-Amsterdam. */
hummingbird..type                   /* 'zoo' is the type. */
hummingbird..name                   /* Artis-Amsterdam. */
hummingbird.location.location       /* Amsterdam. */
```

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 demons, 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. Their definition is evaluated upon initialization of the guardian. 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 their specification (and global optimization) the common part of the rule constraints is factored out and placed at the beginning. Third, the guardian definition terminates with an optional access constraint which regulates access to the guardian properties.

The statement blocks associated with guardian rules are interpreted as transactions, i.e. atomic, durable, and consistent. 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. It is assumed that the transactions are coordinated by a transaction manager.

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 to objects with the attribute isa-bird, 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 is 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. Its value is accessible from any bird object 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) [
         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. A variable is bound by an assigned of a reference to an object through an object selection expression. An alternative binding of variables 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 clause and set construct. The set constructor can also be used to declaratively select object sets for modification, i.e. they may 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.isa-bird                                          /* binds with some bird */
(Bird: Bird.isa-bird and Bird.food= 'honey')   /* binds with exactly honey eating bird */
{Bird: Bird.isa-bird and Bird.food= 'honey'}   /* binds with all honey eating birds */
{Bird: Bird.isa-bird and Bird.food= 'honey'}:+ sale    /* all honey eating birds are for sale */
```

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, repetitive 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 an overloaded function will be based on the scope expression belonging to the function. So function selection is based on instances of the arguments. 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)
]
```

3. Database evolution

In this section we illustrate how a database can be described in *Godel* and how the evolution of the database can be controlled by the *guardians*. First, we review the notions on database integrity and show that an important class of integrity rule has been omitted in the traditional systems. Second, we present a series of examples which illustrate the use of guardians in controlling the contents of a relational-like database. Finally, database evolution is exemplified by mapping some *evolutionary* guidelines into guardian definitions.

4. Database Integrity.

Some advocates of the relational data model define a database as an interpretation of a first order theory, where the non-logical axioms are general laws described the perceived view on the world.[10] In this view the database states are governed by assertional statements, describing invariants over the successive database states. These assertions are either given in terms of a single database state, i.e. a *state rule*, or described in terms of a transition invariant, i.e. *transition rules*.

In practice, however, description of database integrity is severely limited due to the lack of functionality in most database management systems. Functionality must be provided at three levels; data definition, data manipulation, and the application programming language. Along with the definition of a table one can specify value ranges, referential constraints, and null constraints.

One way to enforce integrity is through a database trigger mechanism. A database trigger is an action activated as a side-effect on user directed modifications on the database. Each trigger consists of an event specification, an assertional statement and an action. The triggering schemes implemented fall short in functionality. A transaction can only be undone and a simple message can be given.

The last resort to ensuring database integrity, as described by the assertional rules, is to (automatically) embed them into the application programs. Thus, a high-level programming language, such as Pascal or Ada, is used to alleviate the shortcomings of the data model and database system. Although these languages allow complex conditions to be integrity laws to be maintained, their shortcomings stem from the specification rigidity . It is difficult to amend a complex information system described by a series of Pascal programs to reflect a new integrity constraint. Moreover, the languages are less suited to do both algorithmic and symbolic processing at the same time (as is needed in a DBMS).

An important class of integrity rules has escaped the attention of most researchers working on the theory and implementation of database integrity. Namely, integrity rules which work in the face of exceptions and which require a particular state property to become true in some derivable state. Such rules will be called *evolutionary rules.*

To illustrate, consider a employee database of a university and a new computer scientist being inserted into the database on the day he is hired. Then it should also be ensured that he receives a contract within 3 months thereafter. In case this guideline is violated then the database system should take active measures to ensure that this guideline is satisfied as soon as possible. For example, by reminding the head of personnel department (or his substitute) about the violation. Possibly stimulating and engaging in a dialogue between head of Personnel Dept. and head of Computer Science Dept. (Guided electronic conferencing)

A solution to the old integrity problems and the evolutionary rules is given by the *guardian* concept in *Godel*, since it combines the declarative power of a predicate-based definition of database integrity (by expressing a state of interest) and the algorithmic power to efficiently deal with unwanted situations in an effective way. Moreover, the transaction concept and computational model underlying the language aids in the descriptions of evolving database. In the following sections we show how these claims are substantiated.

4.1. Integrity preservation through guardians

As mentioned in chapter 2 the *Godel* database consists of a large set of named objects, each having a set of attributes. Attributes take value sets and references to other objects in the database. Thus, a prototypical object description is shown below.

examples

```
object employee [
        isa-employee
        ename := 'Jones'
        address := 'Park Lane 1'
        city := 'Amsterdam'
        age := 32
        hiringdate := today
]
```

In this description we can recognize the structure of a tuple in the relational database world. This means that we need to ensure that all attributes of an employee tuple are present and that simple value constraint (age>0 and age<110) are obeyed. The *guardian* to implement these constraints is shown below. The guardian limits the state space by only considering tuples which have at least an attribute saying that it represents an employee. For those objects it checks for superfluous attributes and reports on them to the user. We need not remove these superfluous attributes, because it is allowed in *Godel* to associated with the same object description additional attributes describing a different role of the entity it represents.

examples

```
guardian employee-class
when O.isa-employee
[
        when {O.A.name}- {employee, age, ename, address, manager,hiringdate} <> {}
        [
                Garbage := {O.A}- {employee, age, ename, address, manager,hiringdate}

                do Garbage.E [
                        write [E.name, ' is an illegal employee attribute\n']
                ]
        ]
        when O.ename.self = undef [  O :+ ename ]
        when O.age.self = undef [  O :+ age ]
        /* etc.. */

        /* range constraints */
        when O.age<0 or O.age>110 [
                write [O.name,' Illegal age value\n']
                O.age := O.age.oldvalue
        ]
]
```

Notice that our interpretation of guardians as an independent transaction leads to a time slot where the user has updated the database i.e. committed the transactions and this guardian sees the effect. That is, enforcement of these integrity constraints are not part of the transaction the user initially fired. To ensure these rules as part of the original transaction requires an alternative specification, using the function primitives. This however, is beyond the scope of this paper.

The three most important integrity rules enforced by a relational system are tuple uniqueness, atomicity of the attribute values, and the enforcement of referential constraints. All three can be described in a generic way using a single guardian; as shown below.

examples

```
        object relations [employee ]
        object employee [ age ename address manager hiringdate]

        guardian relational-constraints
        when T.Relname = relations.Relname [

                /* attribute value atomicity */
                when Relname.A and count( {A.value: T.A} ) > 1
                [
                        write[T.name, T.A.name, ' violates atomicity property\n']
                        T.A := undef
                ]

                /* tuple uniqueness */
                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']
                ]

                /* referential integrity */
                when T.manager = undef or {M : M.ename = T.manager} = {}
                [
                        write [T.ename, ' has unknown manager\n']
                ]
        ]
```

The database schema is described by two objects types. One for the relations and one for each relation describing its attributes. These objects are similar to the meta-relations in commercial systems.

The guardian rules are split into three sections. This first section checks for all attributes of an object its atomicity. As soon as a violation is found a warning is issued and the attribute value is set to undefined. Note, that the erroneous values remain accessible through the history mechanism defined for attribute values. The second rule tries to locate a different object in the database which is identical at the value level. For this purpose we rely on a function equaltuple, defined outside elsewhere. The last rule illustrates how a specific referential constraint is described. A generic description of referential constraints is left as an exercise.

The next example illustrates the solution to the new scientist problem, i.e. how can we describe the guideline that the scientist should have a contract within three months after his hiring date. This is implemented by the following guardian. As soon as the guideline is violated it modifies the agenda of the head-of-personnel department to look into the problem. Note, that this action does not force him to take any step. Rather it helps reminding him the problem. Moreover, to avoid repetitive modification of the agenda the rule ensures that this point has not been added already.

examples

```
guardian contracts
when O.isa-employee and not O.contract
[
        /* ensure that contracts are delivered in time */
        when O.hiringdate+ months(3) < today  and
                not head-of-personnel.agenda..O
        [
                head-of-personnel.agenda :+
                        object[
                                contract
                                topic := 'no contract yet'
                                referent := O.self
                        ]
        ]

        /* Ensure that new employees get a prepayment */
        when O.salary = undef and
                O.prepayment = undef and
                O.prepayment.reason = undef and
                tomorrow = payday
        [
                ttyout := head-of-personnel.terminal
                write ['What is the prepayment of ',O.ename]
                read O.prepayment
        ]
]
```

The second rule illustrates a more direct action of the database system to ensure that all guidelines are satisfied. It requires that all personnel with an undefined salary the day before pay-day should be inspected and a 'pre-payment' should be established. This check involves a direct communication with the head-of-personnel. As soon as he logs on to the system this guardian will send a message asking for the pre-payment value. The head-of-personnel is forced to answer or provides explanatory information before the system allows him to continue.

The last example shows how the database can be modified on the fly to accommodate a new attribute called function level. The function level is assumed to be implemented by a discriminative routine described elsewhere. Note that the function level need not be implemented as a single batch job. Objects can be handled one at a time, i.e. each modification is a transaction. Moreover, this guardian ensures that the attribute is automatically incorporated for new employees in case the head-of-personnel forget it to be mentioned.

examples

```
guardian convert-database
when O.functionlevel = undef
[
        when true[
                O :+ functionlevel          /* add attribute */
                O.functionlevel := classify(O)
        ]
]
guardian convert-database
when O.dfl or O.usdollar
[
        when O.dfl and O.value [
                O :- dfl
                O :+ ecu
                O.value := O.value / 2.50
        ]
        when O.usdollar [
                O :- usdollar
                O :+ ecu
        ]
]
```

Included in this example is an automatic conversion of DFL and USD into ECU. In combination with overloaded arithmetic functions it enforces a uniform format of object representations, while at the same time supporting a mixed use. This is necessary, because the conversion guardian may not have been finished when the value of a USD contract us manipulated.

5. Reflections

In this paper we have introduced a new approach to information systems development. Our approach is centered around the concept of a *guardian*, a warder of the database, as embedded in the object-centered database language *Godel*. The object-centered paradigm provides the flexibility to model real-world entities in all its forms, without the constraining typing mechanism found in other approaches. In particular, we have indicated that concepts such as inheritance can be modelled in *Godel* to accommodate any of its semantic interpretations.

We have shown that the guardian programming paradigm can be used to effectively deal with both the traditional integrity rules and the evolutionary rules. The latter we have demonstrated by using automatic transactions to maintain not only the database integrity, but also to model application programs. Thereby turning the database system into an active entity rather then a passive repository of facts.

A functional prototype *Godel* processor has been implemented under BSD 4.2 in C-Prolog. Our initial findings working with this implementation proves the validity of the language features. However, the performance of the examples shown 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. Moreover, a parallel processor architectures should be considered for improved execution speed.

The second area of attention is the language functionality and the programming methods to be used. Predicting the behavior of a *Godel* program is complex, because it is not always clear in advance what the combined, non-deterministic effect is of running the guardians in parallel. Determinism should be encoded in the *Godel* application. A particular area where these aspects are currently being studied is the design of an inference engine within the 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. *International Conference on Expert Database Systems*, 1986.
2. Armstrong, W.W. and Delobel, C., "Decomposition and Functional Dependencies in Relations," *ACM Transactions on Database Systems* **5**(4), pp. 404-430 (Dec 1980).
3. Codd, E.F., "Extending the Relational Model to Capture More Meaning," *ACM Trans. on Database Systems* **4**(4) (December 1979).
4. Elmasri, R., Weeldreyer, J., and Hevner, A., "The category concept: An extension to the entity-relationship model," *Data & Knowledge Engineering* **1**, pp. 75-116 (1985).
5. Gallaire, H., Minker, J., and Nicolas, J.M., "Logic and Database; A deductive Approach," *ACM Computing Surveys* **16**(2), pp. 153-186 (June 1984).
6. Hammer, M.M. and McLeod, D.J., "The Semantic Data Model: A Modelling Mechanism for Database Applications," *Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data.*
7. Kersten, M.L. and Schippers, F.H., "Towards an Object-Centered Database Language," *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, Cal. (Sep 1986).
8. Kersten, M.L. and Schippers, F.H., "*Godel* A General Object-Centered Database Language," CW-8615, Centre for Mathematics and Computer Science (April 1986).
9. Maier, D., Ullman, J.D., and Vardi, M.Y., "On the Foundations of the Universal Relational Model," *ACM Transactions on Database Systems* **9**(2), pp. 283-308 (June 1984).
10. Nicolas, J.M. and Yazdanian, K., "Integrity Checking in Deductive Databases," pp. 325-346 in *Logic and Database*, ed. J. Mincker (August 1978).
11. Smith, J.M. and Smith, D.C.P., "Database Abstractions: Aggregation," *CACM* **20**(6) (June 1977).
12. Smith, J.M. and Smith, D.C.P., "Database Abstractions: Aggregation and Generalization," *ACM Trans. on Database Systems* **2**(2) (June 1977).

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 } .
```