F2: an Evolution Oriented Database System

Th. Estier, G. Falquet, M. Léonard
Centre Universitaire d’Informatique
Université de Genève
rue Général Dufour, 24
1211 Genève 4 — Switzerland
(estier | falquet | leonard)@cui.unige.ch

ABSTRACT. The features of a database system specifically designed to support evolution are presented. The design of the F2 system has been deliberately directed toward integration of evolution features and the flexibility of structures at every level. This has leaded to an architecture where the meta-circular organization of the objects management is not an aesthetic facet but a concrete property of the system.

1 Introduction

The F2 database system was developed from 1989 to 1991 at the University of Geneva. The starting point of the project was to consider the evolution of schemes and objects as an inherent and fundamental property of a system rather than a “later-added” functionality. The preceding experiences of the research group on semantic data models and information systems design, have clearly shown that evolution could not be well integrated into the underlying database system by simply adding a set of features on top of it. Quoting Lohman & al. in about extensibility, we state that evolution “cannot be retrofitted; it must be a fundamental goal and permeate every aspect of the system’s design”. This idea has been adopted as a major principle that motivated design decisions at every level, including the data model, the central role of a dictionary, the object manipulation primitives and triggers, the view mechanism, and the object storage management.

Data model

The F2 model is both influenced by the “object-based” models and functional models (like Z or IFO). It was built around three concepts: classes, attributes and specializations. Operations on objects are defined using four primitives: Create, Enter, Update & Delete. These atomic actions are defined such that referential integrity and subclassing consistency are permanently maintained.

The set of concepts used in this model has been kept as small as possible. There is no explicit type constructor like set_of or tuple. Sets of objects are built by multi valued attributes and tuples are naturally formed by the set of attributes attached to a class. Structures for complex objects consist of combinations of classes and attributes. This small set of components made easier the

1 The project was sponsored by Swiss-FNRS under grant #1603-0.87.
definition of the model by itself, which was essential to elaborate the dictionary and to actually build the system around it.

**Dictionary**

The F2 dictionary is the set of classes that describe base schemas, like the meta-level of some object-oriented database systems or the catalog of relational systems. But in the case of F2, there is no real border between objects and meta-objects. This set of classes is not limited to a documentary catalog describing existing bases, it is the only way to create and manipulate such bases. Concretely, this means that creating a class in F2 is obtained by applying the `Create` primitive to the class `CLASS`. And so on for attributes, subclasses, etc. There is no need for a schema definition language, and any change to a schema is formulated only in terms of primitives applied to dictionary classes.

This implies two things: first, the initial set of objects could not be elaborated by the F2 system itself and was obtained by a bootstrap process\(^1\). Second, the semantic of primitives must be extended for specific classes by a trigger mechanism. When a F2 system starts up for the first time, the object base is not empty, it already contains a root base that will be extended and modified by the creation of new object bases.

**Triggers**

When one of the primitives is applied to an object, it is possible to execute further methods which are specific to the concerned class. This co-execution is possible for each primitive at two points: right before their execution (for additional pre-conditions) and immediately after (to extend their effect on the database). These points are called triggers and are stored in the dictionary as an F2 class of objects (e.g. `pre_Create`, `post_Create`, `pre_Enter`, `post_Enter`, etc.).

For example, the creation or modification of an object of the class `Attribute` executes strictly the same primitive as for any other class, but then post-triggers a specific method that allocates resources in the storage system. The association of triggers with pre- and post-methods is simply controlled by manipulating objects of classes `Trigger` and `TriggerType`. This gives a powerful way of defining repercussions of schema objects changes on their instances.

**Objects storage**

In many object-oriented dbms (e.g. Gemstone [7] or O₂ [9]) the storage model is “object-oriented”, each logical objects is simply mapped to a physical storage object (record) which is a container for the objects’ value. The storage model of F2 objects is radically different, it is “function oriented” or “transposed”. Instead of containing all the values of an object, each storage object contains all the values of an attribute, i.e. a function from oids of one class to objects of another.

The clear distinction between an object representation and its storage resources greatly eases the change of structures applied on large collections of objects. Each attribute is implemented by an abstract persistent type called a N-function. All F2 structures, including the dictionary, are stored and retrieved using this unique type of data.

---

\(^1\) pretty much the same way Pascal compilers written in Pascal were bootstrapped twenty years ago [20].
**Modules**

The module concept of F2 has been designed to fulfill two roles: it allows to encapsulate simple and complex objects and, more generally, any logical access structure (not necessarily hierarchical) between sets of objects. It also provides a view mechanism on the logical database schema. Module operations can be defined that act on subsets of such complex encapsulated structures.

This mechanism provides a high degree of independence between the module’s client (a database user or an application program) and the logical database structure. As a consequence, users or applications can access and manipulate complex data structures without having to know how they are represented in terms of classes and attributes. Moreover, modules can isolate users and applications from schema changes.

The following sections of the paper describe in more details the different components of the system for which design decisions were motivated by evolution support.

**2 The F2 model and object manipulation primitives**

A class is both a type (structured or not) and the set of all objects of this type. The type describes the objects structure and the permitted operations. An object can be identified independently of its value through an object identifier (oid). Atomic objects like numbers or character strings are self identifying, i.e. value and oid are equivalent.

An attribute is a multi valued function between two classes. These functions express properties of the objects like $\text{work\_for(}Project'F2) = \{\text{Tom, Jerry}\}$. When the maximum cardinality of an attribute is known to be one, the function is considered mono valued like in $\text{address(}\text{Tom}) = \text{Geneva}$. A class $C$ with attributes $A_1,...,A_n$ is called a tuple-class. The value of an object $o$ of $C$ is the tuple $[A_1(o), \ldots, A_n(o)]$.

A specialization (or subclass) of a class $C$ is another class $SubC$ whose objects all belongs to $C$. In the opposite, the set of attributes of $SubC$ is formed of all the attributes of $C$ (inherited attributes) to which may be added specific attributes. Take for instance a class $\text{Person}$ with two subclasses $\text{Employee}$ and $\text{Customer}$. As $\text{Tom}$ is a member of the class $\text{Person}$, it is correct to invoke both his name and his address. Since he also belongs to the subclass $\text{Employee}$, one can compute (or query for) his salary. As opposite, the $\text{Person’Bill}$ which also belongs to the subclass $\text{Customer}$, has a name, an address and a set of ordered products. Membership of subclasses are not exclusive, like for $\text{Mary}$ who is both an $\text{Employee}$ and a $\text{Customer}$. One can say that she inherits both subclasses attributes. Note that there is no need in this case to explicit the class of the $\text{Persons\-who\-are\-both\-employees\-and\-customers}$.

**Object manipulation primitives**

Operations on objects are defined using four primitives: $\text{Create}$, $\text{Enter}$, $\text{Update}$ & $\text{Delete}$. These atomic actions are defined such that referential integrity and subclassing consistency are permanently maintained. Their respective semantic is briefly described below.

- For a class $C$, $\text{C.Create()}$ returns a new oid which from now on belongs to $C$ and all the superclasses of $C$. 
• Applying \textit{SC.Enter}(o) means that \(o\) must be considered as a new member of subclass SC, while it already existed in one of the superclasses. In other words this specializes object \(o\).

• If \(A\) is an attribute inherited by \(o\), \textit{Update}(o, A, Val) fixes the new value of \(A(o)\) to become \(Val\). This is possible only if \(o\) and \(Val\) satisfy specific conditions: \(Val\) must be a set of oids belonging to the domain class of \(A\) or it may contain the null value ‘?’ and the set size must satisfy the cardinality constraints (min & max) possibly defined on \(A\).

• Applying \textit{C.Delete}(o) means that \(o\) must leave the set of objects that belongs to \(C\). To maintain the specialization consistency, \(o\) also disappears from all the subclasses of \(C\). If \(C\) is a root class (has no superclass), \(o\) will completely disappear from the database. To maintain referential integrity, objects \(o’\) that were referring \(o\) must be modified: the \(o\) oid is removed from every value \(A(o’)\) where \(A\) is an attribute of domain \(C\), and even worst, if \(A(o’)\) becomes smaller than the minimum cardinality stated for \(A\), the deletion algorithm is recursively applied to \(o’\).

The Fig. 1 illustrates an example of how these rules apply. Consider a class \textit{Designer} containing three objects: Tom, Bill and Mary. \textit{Manager} is a subclass of \textit{Designer} and contains just Mary. Project is another class defined on a name, a leader (at least and at most one \textit{Manager}) and a set of members (possibly empty but at most six \textit{Designers}). Now consider the project P26 with Mary as leader and \{Tom, Bill\} as members. Suppose we apply \textit{Designer.Delete}(Tom). Suppressing Tom from this base will cause a modification of \textit{members}(P26) to \{Bill\}, and one is a legal size for sets \textit{Project.members}. Now suppose we execute \textit{Manager.Delete}(Mary). This will remove Mary from class \textit{Manager} but she remains in class \textit{Designer}. This also causes the deletion of project P26 because possessing a leader was defined as mandatory for a project (at least one)\(^1\). Executing \textit{Designer.Delete}(Mary) is also legal since a manager is a designer and would have the same effect plus the complete suppression of object Mary from the database.

\(^1\) note that we could have previously assigned the value ‘?’ to \textit{leader}(P26) to avoid its destruction now.
Every operation (or method) defined on an object is expressed in terms of these primitives. The recursive nature of the Delete primitive tends to be very powerful when applied to dictionary objects to describe schema evolution.

3 Dictionary

The F2 dictionary is an initial set of classes and objects from which the structure of a database is built. The schema of these classes is given in Fig. 2. Nodes stand for classes, arrows for attributes, subclasses are embedded nodes with grey arrows. This graph has been somewhat simplified for clarity. Most of the attributes with domains Int or String are not drawn. A tuple class possess generally at least one attribute (although a tuple class without attribute can exist and have property-less objects). If a class is atomic, its objects have atomic values of a specific base type (like “character strings” or “floating numbers”) and the class is then described by inf- and sup-bounds, or by additional information (like the maximum length of variable strings). An example of atomic class is Boolean, with base type “natural numbers” and bounds (0..1). The classes Module, Arc and Node describe modules structures (see parag. 6). Triggers and methods are explained in parag. 4.

When the F2 systems starts up, the dictionary classes already contain their own description. This initial state of any F2 database is the result of a bootstrap process. In the first stage of this process, only the kernel of the dictionary is built (classes Class and Attribute). Actually, the first program that created the kernel was generated from the output of a spreadsheet! The second step of the bootstrap is executed by a special version of the F2 system that inhibits all constraints validations when primitives are applied. This program builds all the other classes of the dictionary (Modules, Triggers, and so on).
### 3.1 Schema evolution primitives

A schema of classes and modules is created and manipulated by directly applying primitives to the dictionary objects. For example the schema of Fig. 1 is created by the following sequence:

```
des := TupleClass.Create();  -- des is a new tuple class...
Update(des, name, "Designer");  -- ...with name "Designer"...
dn := Attribute.Create();  -- dn is a new attribute...
Update(dn, name, "name");  -- ...with name "name" !...
Update(dn, from, des);  -- ...attached to our class des...
Update(dn, domain, Class'String);  -- ...with class "String" as domain...
<etc...>
```

The above sequence is written in FarTalk language, but the same primitives are available in Ada, the present F2 host language (see parag. ). Modifications in any part of a schema is made by combinations of the same primitives. As the manipulated objects are in fact classes or related to classes definitions, it is necessary to extend their side effect to the objects of their extensions. In other words, the effect of schema alterations must be propagated along “instance_of” links.

For example, given a subclass SC, if one want to assign a new superClass(SC), the sole validation that the assigned value should refer to an object of the class Class, is not sufficient. The new superclass must also belong to the same hierarchy of specializations as the previous one. An additional pre-condition must then be taken into account for execution of superClass.Update().

The post-condition (or side-effect) of this assignment should also be extended: the partial order on subclasses inclusions must be preserved in the current specialization tree. A generic mechanism for extending both pre- and post-conditions of a primitive application has been defined: the internal triggers, described in parag. 4.

### 3.2 Extending the model

The dictionary can easily be extended to introduce additional concepts to the model. For example, the class Key has been added to introduce the concept of identifying set of values in a class. A key is a set of attributes that identifies at most one object of this class. This introduces a uniqueness constraint on the concerned class for each defined key which must be validated by any value assignment to objects of this class. This extension was done by adding the schema of Fig. 3 to the dictionary. A key may be defined on any direct or inherited attribute of the class. This

![Fig. 3 introduction of keys in the F2 system.](image-url)
means that an attribute can restrict its identifying role to a subclass only (uniqueness is garantied only among objects of the subclass). It has then been possible to define keys on existing classes: a class can be identified by its name, a module node is identified by a pair made from its name and the module it belongs to, and so on1. Validations of keys constraints have been added by generic methods which are then triggered when appropriate primitives are applied to the concerned class. A key may also be set “inactive”, during a particular phase of a transaction and globally validated at the end of this transaction, after its re-activation.

In addition to its integrity checking use, the concept of key has proved to be very useful for the definition of the FarTalk language (see parag. ). It restricts the need for persistent “names” or “symbols” like in SmallTalk for example.

4 Triggers

A trigger is an internal event which associates the execution of a primitive on a specific class to other methods. There are two kind of triggers: those who cause additional pre-condition to be checked right before the primitive execution and those who cause additional post-conditions or side-effects immediately after the primitive (see Fig. 4 ). Each primitive has a trigger type of each kind: pre-Create, post-Create, pre-Enter, post-Enter, pre-Update, post-Update, pre-Delete, and post-Delete. This mechanism is particularly useful to implement and monitor propagations of dictionary objects updates. The triggers are the concrete link existing in F2 between a class and its objects or more generally, between a meta-object and its derived objects.

4.1 Example of triggers activation: creation of a class

The following example describes what happens during the creation of a TupleClass. Suppose the system executes: NewC := TupleClass.Create(). As the new object will belong to both classes Class and TupleClass, triggers for both these classes are considered. In this case, there is no pre-create trigger. Then the create algorithm elaborates a new oid. The post-trigger of TupleClass

---

1 the key of the class Key is the pair of attributes (ofClass, keyAttribs).
is associated to a method that creates an invisible state attribute\(^1\), by simply applying the primitive \texttt{state\_Attr := Attribute.Create( )}. This will then trigger the post-method for attributes creation, i.e. the allocation of new resources in the storage management level for the underlying N-Function.

### 4.2 When objects of a class are classes themselves

The trigger mechanism and the dictionary can be used to generate multiple levels of instantiation links. In other words, it is possible to model a class with any specific attribute such that each object is itself another class. Let’s take for example a class of cars where each car stands in fact for a class of car units (see Fig. 5). We suppose here that these later classes all have distinct structures (i.e. specific attributes). To model this in the F2 system it is sufficient to modify the class \texttt{Car} so that it becomes a subclass of the class \texttt{TupleClass}. By this way, each car inherits the attributes and behaviour of tuple classes. In particular, each creation of a car will also trigger methods specific to tuple classes (as described above), hence will allocate storage resources for containing specific objects. This two-level modeling allows to distinguish between classes properties and instances properties (or attributes). Of course, nothing prevents from reiterating the process to any level. The difficulty may then rely in a possible lack of convincing examples that would need more than 2 or 3 successive cascades!

This example illustrates that it is not easy to identify a clear frontier between a meta-level and an object-level in both the F2 model and the F2 system. Experiences led in the field of terminology and concepts databases [15] showed that it is preferable to consider that there is only one level of objects, and that links between objects and classes express only a specific semantic link.

\(^1\) which will be used to store objects states. see parag. 5
5 Objects storage and representation

In many object-oriented dbms (Gemstone [7], O₂ [9]) the storage model is “object-oriented”, each logical objects is simply mapped to a physical storage object (record) which is a container for the objects’ values. The storage model of F2 objects is radically different, it is “function oriented” or “transposed”. Instead of containing all the values of an object, each storage object contains all the values of an attribute, i.e. a function from oids of one class to objects of another.

5.1 Attributes and persistent functions

The oid of an object of a tuple class is a pair (class-hierarchy-id, object-number), thus an attribute to a tuple class can be represented by a function from the positive integers to the positive integers (or to subsets of the positive integers if multivalued). On the other hand, attributes to atomic classes, whose objects are self identifying, are represented by functions from the positive integers to (subsets of) strings, real, integers, boolean, depending on the base type of the atomic class. The F2 physical storage manager stores and manipulates these functions, called N-functions, as persistent objects. In the current implementation, N-functions are stored as variable size persistent byte arrays, hence oids can be used as index values to find attribute values in the corresponding byte array.

In addition to the attributes explicitly defined in its class schema, each class C is assigned a “state attribute” named state_C whose value determines the current status of each (potential) object. When a new object is created in class C, the N-function corresponding to state_C is examined to find an integer n such that state_C(n) is undefined, this value is selected as the oid of the new object and state_C(n) is set to “exists”. At the same time, for each attribute A of C, the corresponding N-function, say r, is updated to set r(n) to the value ? representing the “unknown” value.

5.2 Subclasses

In F2, the inclusion for subclasses extents is enforced by the physical representation of objects. When an object is created by primitive n := C.Create(), it is first considered as a member of the root class RC of C (the class at the top of the subclass hierarchy to which C belongs). The creation method described above is applied to RC and yields a new oid n for this object. Then, in each superclass SuperC of C, state_SuperC(n) is set to “exists”, as well as state_C(n).

Applying the primitive SubC.Enter(n) in a subclass SubC amounts to set state_SubC(n) to “exists”, as well as all the state_x(n) of the superclasses of SubC. Applying SubC.Delete(n) from the same subclass SubC amounts to set state_SubC(n) to “removed”, as well as all the state_X(n) for the subclasses of SubC.

The state attribute also act as a tag to check if an attribute currently applies to an object. If one want to access to attribute A of object o, A must be an attribute of one of the classes to which o currently belongs. This can be checked by 1) finding the class C of the hierarchy under the root class of o in which A is defined1 and 2) verifying that state_C(o) = “exists”.

Fig. 6 illustrates the transposed organization of data in the storage management level.

\[1 \text{This class is unique since F2 does not allow redefinition of an attribute in the same class hierarchy}\]
5.3 Support for schema evolution

Object storage through independent functions is obviously well suited for supporting the addition and removal of attributes during database operation. Adding a new attribute to a class amounts to ask the physical storage manager to create a new persistent function, none of the existing functions require changes. This contrasts with the object based storage where adding an attribute requires to make room for its value in each object container, often causing physical objects moving, storage reallocation and low level pointers updates.

For the same reason, changing the values of an attribute \( A \) by applying a function \( f \) to them is easily supported, even if the application of \( f \) changes the domain of \( A \). For instance consider the following two classes:

```
class Project (name: string, team: Team, ...);
class Team(members: Employee * (2..10), leader: Employee, meeting_point: Bar);
```

We can decide to replace each value \( t \) of attribute \( \text{team} \) by \( \text{members}(t) \), this turns attribute \( \text{team} \) to a multivalued attribute with domain \( \text{Employee} \).

At storage level this operation is carried out by 1) creating a new N-function \( nf_{\text{team}} \); 2) storing in \( nf_{\text{team}} \) the value \( \text{members}((\text{team}(p)) \) for each \( p \) in \( \text{Project} \); 3) assigning \( nf_{\text{team}} \) to attribute \( \text{team} \) and deleting the previously assigned N-function.

5.4 Performance issues

The attribute-oriented F2 storage model, that physically clusters attribute values instead of objects values, has several interesting properties:

1. Some processes need to access the value of a single attribute in each object of a class.

   For instance, to compute aggregate functions (sum, average, etc.) on one attribute or to do pattern matching retrieval. With “o-o” storage, reading the value of attribute \( A \)
on object \( o \) requires the transfer of the whole object in main memory, thus the above cited processes reads all the information about all the objects of a class. If the storage space used for the attribute of interest represents say 10 percent of the space occupied by an object, 90 percent of the time is spent reading useless data. N-functions storage allow to access the attribute’s values of interest and only those ones.

2. Computing joins between several classes, only those attributes involved in the join are brought into main memory. Since these attributes link tuple classes to tuple classes, they are compactly stored as oids arrays which can fit in main memory, even for databases with tens of thousands objects. In many cases this enables to efficiently compute joins on non indexed attributes.

6 Modules

At the “first” level, F2 objects are not encapsulated. Attributes of F2 object are always visible and updatable from the outside. Similarly, complex objects must be represented by several classes connected by attributes representing “part-of” links. So far complex objects cannot be encapsulated.

The module concept of F2 has precisely been designed to provide a high degree of independence between the module’s client (a database user or an application program) and the logical database structure [13]. Usually this kind of logical independence is supplied by a view mechanism. However, views generally lack a general update capabilities [8]. Views defined by join expressions represent one logical access path between all their components, if one is interested in only a subset of the joined objects/classes one has to define a new view.

In OO systems, defining views as virtual classes pose new problems. If a view is a virtual class then oid’s must be provided (invented) for its virtual objects. In general, nothing guarantees that successive computations of the same virtual class will yield the same oid’s [1], unless oid’s values are based on identifying attribute values of objects [17]. Another important issue deals with the position of the computed virtual classes in the database class hierarchy [1], [19]. These yet partially unsolved problems lead us to define a module as an abstraction that, instead of creating new virtual objects, computes associations between existing objects and represents them as tuples of objects (o-tuples).

6.1 Structure and semantic of a module

A database module is an associative abstraction over database objects. The data semantics of a module is given by a set of tuples, the components of which are objects. Thus, the semantic function of a module, called the connection function, maps a database state to a set of object tuples (o-tuples). This function is entirely specified by the module schema which is a directed graph composed of:

- A set of nodes, each node represents a set of objects of the same class/type.
- A set of arcs, each arc represents the connection of objects of the source node to objects of the destination node through an attribute.
For a set $X$ of nodes, the *connection graph* on $X$ is the subgraph of the module’s structure that contains all the simple, non-directed, paths between all the pairs of nodes in $X$. The connection on $X$ is defined as the join according to the connection graph, eventually projected on $X$.

**Example 1** In module $M_{Projects\_Skills}$, the connection on $\{E, P\}$ is composed of the object pairs $[E; e, P; p]$ such that $e$ is employed in a department $d$ that has project $p$ and $e$ possesses a skill $s$ that is required by project $p$. Here the join graph is the whole module structure since it must contain the two paths: $(E \text{ employees } D, D \text{ projects } P)$ and $(E \text{ skills } S, S \text{ required_skills } P)$.

Similarly, the connection on $\{D, P\}$ is made of all the pairs $[D; d, P; p]$ such that $d$ has project $p$ and $d$ employs (at least one) employee $e$ who possesses a skill $s$ required by $p$.

Let us consider now a second module $M_{Departments}$ build on the same database classes and attributes but with a different schema. In this case a department is linked to two sets of skills: the skills of its employees and those required by its projects. The connection on $\{D, E, P\}$ is computed as the join along the path $(E \text{ employees } D, D \text{ projects } P)$ (there aren’t any other path between $D, E$ or $P$ in $M_{Department}$), it gives all the triples $[D; d, P; p, E; e]$ such that department $d$ has project $p$ and employs $e$. There are two distinct ways to associate departments and skills in this modules: the connection on $\{D, P_S\}$ associates departments with the skills required by their projects; the connection on $\{D, E_S\}$ links departments with the skills of their employees.
6.2 Primitive Operations

1. Associating and unifying objects

The purely associative nature of the connection function makes database modules fully updateable, i.e. it is possible to make a new object tuple appear in a connection (Insert) or disappear from a connection (Remove).

The semantic of these operations follows two principles: 1) minimize the number of database changes and 2) avoid non-deterministic behaviour.

Example 2 To insert the object tuple \([D: d, E_S: s]\) in the connection \((D, E_S)\) of \(M_{\text{Departments}}\) an intermediate object \(e\) is created in \(E\), then \(e\) is added to the value of \(\text{employees}(d)\) and \(\text{skills}(e)\) is set to \(s\).

![Fig. 9 Database state after Insert\([D: d, E_S: s]\) into \(M_{\text{Department}}\)](image)

This corresponds to the fact that within this module semantics “adding a skill to the employees’ skills of a department” can be accomplished by “hiring a new employee who has that skill”. There is another way to carry out this operation, namely, to add skill \(s\) to an employee \(e’\) of department \(d\). However, this implies that the system arbitrarily select an employee among the department’s employees and assign this new skill to him. The introduction of such a non-deterministic behaviour is certainly not suitable in a general purpose dbms (principle 2).

From a knowledge acquisition point of view, intermediate objects created by Insert operations represent uncertain knowledge. For instance, in the previous example Insert\([D: d, E_S: s]\) adds the fact that “we know that an employee of \(d\) has skill \(s\) but we don’t know yet the identity of this employee”. This missing information may become available afterwards, when this happens the intermediate object must be unified with an already existing one. A Unify operation has been introduced for this purpose.

Given two objects \(o_1\) and \(o_2\), the unification of \(o_1\) with \(o_2\) replaces every attribute value \(A(o_1)\) by \(A(o_1) \cup A(o_2)\), then it redirects to \(o_1\) every reference to \(o_2\), and deletes \(o_2\).

2. Removing associations

As for Insert, the same two principles define the semantics of the Remove operation. For instance, on the database state shown in Fig. 9, removing \([D: d, E_S: s_6]\) from \(M_{\text{Departments}}\) leads to 1) remove \(s_6\) from \(\text{skills}(e_1)\) and \(\text{skills}(e_6)\) and 2) remove \(e_1\) and \(e_3\) from \(\text{employees}(d)\). Once again this is the only way to make \([D: d, E_S: s_6]\) disappear from the connection on \((D, E)\) without making any arbitrary choice (principle 2).
6.3 Module operations

Operations defined at module level, as opposed to class level operations, do not act on individual objects but on tuples of associated objects. The ability to deal with associated objects is particularly important in a database context. In fact, database processes and transactions often deal with objects which are somehow connected (or to be connected). The implementation of module operations is based on class operations such as Create, Delete, Update, Enter, etc. or on the Insert and Remove primitive module operations. In order to maintain data consistency these operations should not directly update attribute corresponding to arcs of the module structure but instead call the module’s Insert and Remove operations.

Example 3 The following operation could be defined to hire a new employee in a department:

```f2
module procedure M_Departments.Hire(d: Department, e_name: string, e_salary: integer) is
begin
    e := Employee.Create(name: e_name, salary: e_salary); -- create a new object
    Insert[D: d, E: e]; -- and connect it to d
end Hire;
```

```f2
M_Departments.Hire(toys, “John”, 23000) -- hire John in the toys department
```

6.4 Modules and evolution

The logical independence provided by database modules turns out to be particularly interesting when dealing with schema updates. In many cases modules can hide schema updates to users or applications. For instance, consider the schema update consisting in replacing

```f2
class Project(name: string, employees: Employee * (1..50), …)
```

by the two classes

```f2
class Project (name: string, teams: Team * (1..5), …);
class Team(members: Employee * (2..10), leader: Employee, meeting_point: Bar);
```

This update can be hidden to clients of module M_Departments by a structure adaptation consisting in replacing arc (P employees E) by two arcs: (P teams T), (T members E). This way, the semantics of all connection involving P and E are preserved, although they are now computed differently.

7 The Fartalk Language

Fartalk is a simple query and manipulation language, which is fully described in [11]. We will limit ourselves to the informal presentation of several specific features of the language.

**Objects identification by values**

Although objects possess oid’s which are used to designate them, it is often convenient and sometimes necessary to identify objects by their current attributes values. This is particularly true when dealing with objects representing real world entities like persons, cars, colors, etc. In Fartalk a tuple of values can appear wherever an object identifier is expected. For instance, in

```f2
create Department [ name: “toys” manager: [first_name: “Calvin” last_name: “Hobbes”] ]
```
the tuple \[ \text{first_name: “John” last_name “Hobbes”} \] designates the object of class Employee having first name John and last name Hobbes. Of course, this expression is legal only if these values select no more than one employee. When a set of objects is expected, tuples with relational operators can be employed, as in

\[
\text{Employees[} \text{salary < 10000 spoken_languages contains \{“German” “Spanish”\}}.\]

Note here that the tuple may be qualified by the class name when the syntactic context does not specify from which class objects are to be selected.

When a class has a key formed of only one attribute, its objects can be identified by simply giving this attribute’s value. For instance, if attribute name is a key of class CarMaker, the expression “Ferrari” identifies the object of CarMaker having name = “Ferrari” wherever an object of this class is expected. For example

\[
\text{whitelItalianCars := Car[} \text{color = “white” maker in \{“Ferrari”, “Fiat”, “Lancia”, “Alfa”\}.}\]

This example shows how queries involving several object selection/designation may be concisely expressed.

*Queries involving dictionary objects*

Dictionary classes may be queried or updated as any ordinary class and appear in any Fartalk expression. For example, the following expression checks if class MyClass has an attribute price before doing a selection on this attribute.

\[
\text{if Attribute[} \text{from = “MyClass”, name = “price”} = \{\} \text{ then}
\]

\[
\text{myFreeObjects := MyClass[]} \quad \text{-- select all the objects of MyClass}
\]

\[
\text{else}
\]

\[
\text{myFreeObjects := MyClass[} \text{price = 0} \text{];} \quad \text{-- select objects with price equal to zero}
\]

\[
\text{end if;}
\]

Obviously this kind of expression cannot be statically typed checked. This is why Fartalk does only dynamic type checking.

*Other features*

- Modules connections computations and updates, as described in parag. 6.
- Fartalk expressions denote functions which can be composed to form new functions.
- Collections of objects can be stored in variables and manipulated by set operators.

### 8 Implementation Considerations

The F2 system is made of three layers. The lowest layer (or kernel) implements N-functions as a persistent type of data. This encapsulates the paging and block buffering mechanism and a very simplified transaction mechanism (commit / rollback). The second layer (or F2 machine) implements the primitives algorithm, the connection algorithms of the modules and provides a programming interface to F2. The third layer is the FarTalk interpreter which provides traditional facilities for adhoc manipulations and queries.
The whole code (about 25'000 lines) was written in Ada, using exclusively standard library packages. The choice of Ada proved to be judicious for at least two reasons: language features for genericity have been extensively used in both the kernel and the F2 machine and eased the packages structure design, and second, the code was found very easily portable on any platform with a correct Ada compiler.

An interface between the FarTalk interpreter and a set of HyperCard™ commands has been implemented and permitted development of graphical front-ends for F2 on Macintosh™ computers: a graph-oriented editor for F2 schemas (MF2, EcrinsDesign [4]), and an hypertext-based query interface to browse and retrieve objects through modules structures (QFE [11]) \(^1\). A third prototype for a visual monitor of schema evolution and restructuration methods application is under development.

9 Conclusion
The F2 system is a good testbed for evaluation of design choices supporting evolution facilities. The schema evolution primitives have already extensively been used to develop a terminological information system. This last project gave rise to an extension of both the F2 model and the dictionary (see F2\textsuperscript{concept} [15]). We are currently working on incorporating objects and modules versions in the F2 system.

References


\(^1\) HyperCard and Macintosh are trade marks of Apple Computer Inc.
F2: an Evolution Oriented Database System


