Schema Modifications in the LISPO_2 Persistent Object-Oriented Language

Gilles Barbedette
Altaïr
BP105
78153 Le Chesnay Cedex
email: gilles@bdblues.altair.fr

Abstract

This paper addresses the issue of schema evolution in LISPO_2, a persistent object-oriented language. It introduces the schema modifications supported by the LISPO_2 programming environment and presents the potential inconsistencies resulting from these modifications at the schema, method and object levels. Furthermore, it describes how the environment efficiently detects such inconsistencies using a database representing the schema definition. Moreover for correct modifications, it presents how this database is used to update the schema, to trigger method recompilations and to restructure objects using a semi-lazy evolution policy.

1 Introduction

Advanced application domains such as Computer-Aided Software Engineering or Office Automation require both modeling power to represent and manipulate complex objects (e.g. programs, documents or rules) and persistence facilities to store and share these objects between application executions. These new applications led to the development of object-oriented database systems (e.g. [Banc88], [Bane87a]) and persistent programming languages (e.g. [Agra89], [Atki81]). The former augment database systems with expressive power while the latter extend programming languages with persistence. Following this same trend, we developed LISPO_2 [Barb90], a language extending Lisp with the O_2 object-oriented data model [Lécl89a] and orthogonal persistence.

However, a language, alone, does not create a productive environment. The programmer needs tools which deal with the interactive design and implementation of applications. Recognizing the iterative nature of software development, as in [Booc90] and [Goss90], the LISPO_2 programming environment supports an "evolutionary prototyping" development process where design and implementation are not seen as sequential stages but as interleaved ones. In this process, the prototype iteratively evolves from its first version to the final product as the programmer gains experience with the application and refines its design and implementation. In order to support this mode of development, the programming environment has to facilitate the modification of the design in order to incorporate the results of previous experiments. To fulfill this requirement, the LISPO_2

programming environment provides a mechanism for class modification which enables the programmer to change class definitions on the fly, even though some objects have been previously created and some methods compiled. In such a situation, classical environments [Meye88], [Stro86] require exiting the environment, modifying class definitions, recompiling relevant classes and methods and reloading them. Moreover, the test database has to be regenerated. In contrast, the $LISPO_2$ environment checks the consistency of the modifications with respect to the static semantics of the language. Furthermore, it assists the developer in understanding the effects of his/her modification by pointing out the affected methods, and it triggers their recompilation. Finally, it updates objects automatically in order to meet their new class definitions.

The remainder of the paper is organized as follows. Section 2 introduces the features of the $LISPO_2$ language necessary for understanding the rest of the paper. Section 3 presents a taxonomy of the schema modifications supported by the programming environment. The next three sections address their repercussions respectively on schema definition, methods and existing objects. Moreover, they detail the implementation choices that we made to efficiently detect inconsistencies in the schema, to trigger method recompilations and to restructure objects after a schema modification. This is followed in Section 7 by a comparison with other related approaches. Finally, we conclude by summarizing the innovative features of the environment and by indicating future plans.

2 Overview of the $LISPO_2$ Language

This section briefly introduces the features of the $LISPO_2$ language relevant to the issue of schema evolution. For a more detailed presentation of $LISPO_2$, the reader is referred to [Barb90].

- **Classes, Types, Operations and Methods.**

  $LISPO_2$ is a class-based object-oriented language. A *class* defines the structure and the behavior of a set of objects called its instances. The structure of an object is defined by a *type*. A type in $LISPO_2$ is either an atomic type (e.g. integer or float) or a complex type built from other types and classes using the tuple, set and list type constructors. Tuple types are used to model aggregation. Set types represent homogeneous collections without duplicates while list types support indexable homogeneous collections. The behavior of an object is defined by a set of *operations*. A class definition introduces only the specification of operations, called *signatures*. A signature includes the name of the operation, the type/class of its arguments (if any) and the type/class of its result. The implementation of an operation is defined by a *method*. Separating the specification of an operation from its implementation allows the programmer to work with a partially implemented application (no method associated with an operation) or to explore alternative implementations of the same operation (several methods

associated with an operation). Figure 1 shows the definition of the PERSON class. Its structure is described by a tuple structured type defining two *attributes*, and its operational interface contains two operations.

```
(defclass PERSON
        (OBJECT)
        (type (tupleof
                (name string)
                (spouse PERSON)))
        (operations
           (name () (return string))
           (set-spouse (PERSON) (return PERSON))))

(defclass CLUB-MEMBER
        (PERSON)
        (type (tupleof
                (entry-date DATE)
                (spouse CLUB-MEMBER)))
        (operations
           (set-spouse (CLUB-MEMBER) (return CLUB-MEMBER)))
        has-extension)
```

Figure 1: PERSON and CLUB-MEMBER classes

- **Inheritance.**

  Classes are related to each other through inheritance links. A class inherits the structure and behavior of its superclasses. In Figure 1, the CLUB-MEMBER class is defined as a subclass of the PERSON class. Inheritance in $LISPO_2$ is based on subtyping and behavior refinement. The type of a subclass must be a subtype of that of its superclass. Figure 2 gives a syntactic definition of subtyping. For a formal description of the O_2 semantics of subtyping, the reader is referred to [Lécl89a]. In the example shown in Figure 1, the subtyping rules imply that the domain of the "spouse" attribute in the CLUB-MEMBER class (i.e. CLUB-MEMBER) must be a subclass of the one specified in the PERSON class (i.e. PERSON).

  In addition, the subclass may extend or redefine the operations defined by its superclass. Operation redefinition occurs when the subclass defines an operation with the same name as one provided by the superclass. In that case, the operation defined in the subclass must have the same number of arguments as the one of its superclass. Moreover, the types of the arguments and result specified in the subclass must be subtypes of those specified in the superclass. This is illustrated in Figure 1 by the "set-spouse" operation in the CLUB-MEMBER class redefining the one defined in the PERSON class.

  $LISPO_2$ supports multiple inheritance, i.e. a class can inherit from several direct superclasses. Multiple inheritance can lead to attribute or operation name conflicts. These conflicts are

. if T is an atomic type
then T' is an atomic type and T = T'.

. if T is set structured, i.e. of the form (**setof** E) (resp. list structured, i.e. (**listof** E)),
then T' is set structured, i.e. of the form (**setof** E') (resp. list structured, i.e. (**listof** E'))
and E' is a subtype of E.

. if T is tuple structured, i.e. of the form (**tupleof** (a_1 TA_1) ... (a_n TA_n))
then T' is tuple structured, i.e. of the form (**tupleof** (a_1 TA'_1) ... (a_m TA'_m))
$m \geq$ n and $\forall$ i $\in$ [1, ..., n] TA'_i is a subtype of TA_i.

. if T is a class,
then T' is a class and T' is a subclass of T.

Figure 2: Subtyping rules asserting that T' is a subtype of T

solved explicitly by the programmer either by choosing which attribute/operation to inherit or by defining a local attribute/operation in the subclass. This is further explained in Section 4.2.

- **Persistence by Reachability.**

  In $LISPO_2$, persistence is orthogonal to the type system, i.e. all $LISPO_2$ data (either objects or pure LISP data such as vectors or cons cells) have equal rights to persist. Moreover, to eliminate the impedance mismatch problem, we introduce persistence in $LISPO_2$ by extension of the usual LISP data lifetime. We allow data to remain alive between program executions by defining a set of persistent roots. At the end of program execution, all data which are directly or indirectly reachable from the persistent roots are made persistent without any programming cost. These persistent roots are *database variables* and *class extensions*. Database variables retain their associated data between application executions. They can be seen as variables belonging to an everlasting scope. A class extension provides the automatic grouping of all instances of a class (i.e. all objects generated by the class and its subclasses) into a set. A class extension is generated by the **has-extension** option in a class definition as for the CLUB-MEMBER class in Figure 1.

  The *schema* of an application consists of the set of database variables and class definitions appearing in its design.

3 Schema Modifications

In this section, we present the schema modifications supported by the $LISPO_2$ environment and we outline how it processes them. These modifications reflect our intention to start with simple but fundamental and useful modifications in order to understand their impacts on the schema,

its implementation and its associated database. These modifications can be roughly divided into three categories:

- **Modifications of the Persistent Roots.**
  This category contains the addition and deletion of database variables and class extensions. The programmer uses them, as needed, to modify the set of objects that could persist.
- **Modifications of the Class Content.**
  This concerns the addition/deletion of an attribute/operation and the modification of its specification (i.e. the domain of an attribute and the signature of an operation). These allow the programmer to complete a class definition as he/she gains experience with the application concept associated with the class.
- **Modifications of the Inheritance Graph.**
  This refers to adding and removing a leaf class as well as adding and removing an inheritance link between a class and a direct superclass. These are the most fundamental of all modifications since they cope with the general architecture of the application (i.e. the concepts introduced and their relationships).

When the programmer issues a schema modification, the environment processes it in several steps. First, it checks that the modification does not lead to a schema violating the static semantics of the language (e.g. subtyping rules). If it does, the modification is rejected. Otherwise, the environment points out to the programmer the set of methods which can be affected by the modification. Depending on the amount of induced change, the programmer can either confirm or cancel the modification. If he/she confirms it, every affected method is recompiled (if necessary) and marked as invalid if the compiler discovers new type errors. Finally, the relevant objects are restructured. The next three sections address in turn the impacts of schema modifications on the schema, methods and objects.

4 Repercussions of Schema Modifications on the Schema

In this section, we discuss the impacts of schema modifications on the schema. We first define the notion of a *valid schema.* A schema is valid if it satisfies the two following properties:

- The inheritance graph is a direct acyclic graph with one root (named the OBJECT class) and without disconnected classes. Moreover, the subtyping and operation redefinition constraints on inheritance are satisfied.
- There is no name conflict: classes and database variables are uniquely named as are operations and attributes in a class.

We now study each modification and point out how it can break the validity of a schema. Each major case of validity violation is illustrated by means of a simple example. Furthermore, we describe the logical updates to the schema induced by correct modifications. The data structures and algorithms used to efficiently detect the violations are then described.

4.1 Persistent Root Modifications

The addition of a database variable (resp. a class extension option) only implies checking name uniqueness (resp. option uniqueness). Removing a database variable (resp. a class extension option) does not affect the validity of the schema.

4.2 Class Content Modifications

Since the schema modifications concerning attributes and operations involve essentially the same checks, we present them in the same section. Moreover, throughout the rest of the paper, we follow the Eiffel [Meye88] terminology where a *feature* represents either an attribute or an operation. Thus, a *feature specification* represents either the domain of an attribute or the signature of an operation.

- **Add a Feature to a Class.**

  First, the class should not already define a feature with the same name. If the class previously

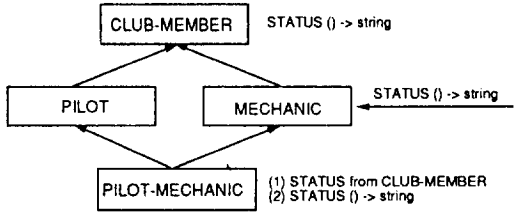

Figure 3: Name conflict in a feature addition

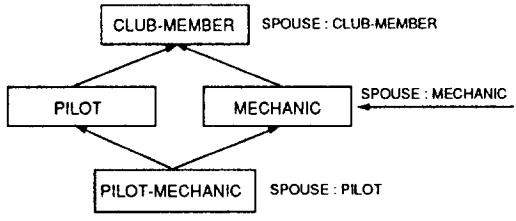

Figure 4: Redefinition error in a feature addition

inherited a feature with the same name, the validity of the induced *upward redefinition* is checked. Moreover, the new feature is propagated to any subclasses. This may lead to both name conflicts and redefinition errors. A name conflict occurs when a subclass

already inherits a feature with the same name but coming from a different superclass. This is illustrated in Figure 3 by the addition of the "status" operation in the MECHANIC class. This addition induces a valid upward redefinition with the "status" operation defined in the CLUB-MEMBER class. However, it leads to a name conflict in the PILOT-MECHANIC class. Indeed, this class inherits both the status operation defined in the CLUB-MEMBER class and the one added to the MECHANIC class. Unlike systems like ORION [Bane87b], $LISPO_2$ does not provide a default rule based on the order of the superclasses to solve such name conflicts. The programmer has to solve them explicitly, either by choosing the operation to inherit using a "from" clause (e.g. the one from the CLUB-MEMBER class as in the first option) or by defining a local operation in the PILOT-MECHANIC class (in the second option). Choosing an operation through a "from" clause does not create a new operation in the PILOT-MECHANIC class. It only points to the operation defined in the CLUB-MEMBER class and thus implies sharing (in particular sharing of the implementation).

Redefinition errors arise when the subclass locally defines a feature with the same name but with a specification that violates the subtyping rules. This is illustrated in Figure 4 where the programmer adds the "spouse" attribute whose domain is the MECHANIC class. This leads to a *downward redefinition* error in the PILOT-MECHANIC class (since the PILOT class is not a subclass of the MECHANIC class).

If there is neither name conflict nor redefinition error, the feature addition is accepted and propagated to every subclass which does not define locally or reference (via a "from" clause) a feature with the same name.

- **Remove a Feature from a Class.**

  Removal of a feature may be performed only on the class defining it. If the feature is referenced through "from" clauses, the modification is rejected. The "from" clauses must first be cancelled (for example by replacing them with local definitions). Otherwise, the only inconsistency that may be introduced is name conflicts in the class. This occurs when the removed feature was previously blocking these conflicts. If there is no such conflict, the deletion is accepted and propagated to every subclass that inherits the feature without redefining it.

- **Change the Specification of a Feature in a Class.**

  This update is only allowed in the class which defines it. The new specification is checked against upward and downward redefinitions of the feature in the class defining it and in every class referencing it by a "from" clause. For example, Figure 5 presents the case where the programmer wants to change the domain of the "spouse" attribute in the MECHANIC class,

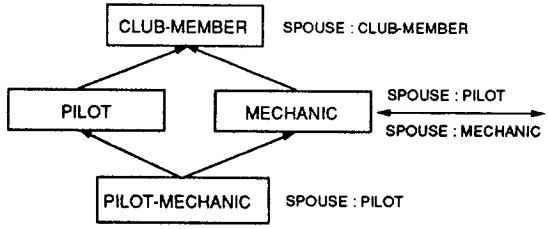

Figure 5: Conflict in feature replacement

stipulating that the spouse of a mechanic must be a mechanic. This modification fails because of the redefinition error occurring in the PILOT-MECHANIC class (i.e. the PILOT class is not a subclass of the MECHANIC class). If there is no redefinition error, the modification is propagated to every class inheriting or referencing the feature.

- **Rename a Feature in a Class.**

  A feature can be renamed only in the class defining it. If the feature is referenced through "from" clauses, the modification fails (since the "from" clause is used to solve name conflicts). The renaming can lead to name conflicts and redefinition errors in the class and its subclasses as for a feature addition. If there is none, the feature is renamed in the class and every subclass inheriting it.

4.3 Inheritance Graph Modification

- **Create a Class.**

  A new class can be created only as a leaf of the inheritance graph. Adding a class in

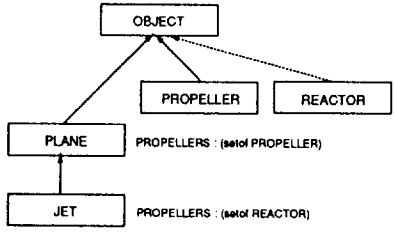

Figure 6: Class creation and the shadow mechanism

the middle of the inheritance graph can be achieved by a combination of class creation and superclass additions. The name of the class must not be used by an already defined class. The superclass(es) specified must have previously been defined. The subtyping and operation redefinition rules are checked. Moreover, if, due to multiple inheritance, a feature name conflict occurs, the programmer has to solve it explicitly.

The innovative facility offered by the LISPO$_2$ environment in this modification concerns the

flexibility in the ordering of class creations. Although the environment requires a class to be created before its subclasses, it does not constrain the classes appearing in the specification of a feature of the new class to be already defined. This allows the programmer to develop and test a design step by step, leaving slices of the inheritance graph undefined while testing others. In such a case, the feature whose specification contains undefined classes (and consequently the class defining or inheriting it) is said to be *shadow*. The inheritance checks involving the undefined classes are presumed correct and memorized by the environment (as described in Section 4.4). However, when a previously undefined class is created by the programmer, the presumed correct checks are then really performed since the position of the class in the inheritance graph is known. If those checks fail, the definition of the previously undefined class is rejected. This is illustrated in Figure 6. In this example, the general PLANE class defines the "propellers" attribute whose domain is the type (**setof** PROPELLER). The JET subclass redefines this attribute with the domain (**setof** REACTOR). However, since the REACTOR class is not yet defined, the JET class and its "propellers" attribute are shadow. The subtyping check (i.e. (**setof** REACTOR) with respect to (**setof** PROPELLER)) leads to checking whether REACTOR is a subclass of PROPELLER, which cannot be performed. Thus, the check is memorized by the environment. When the REACTOR class is defined as a direct subclass of OBJECT, this check is performed completely and leads to a subtyping violation. Hence, the creation of the REACTOR class is rejected and the JET class remains shadow.

- **Delete a Class.**

  This modification can only be applied to the leaves of the inheritance graph. Class deletion in the middle of the inheritance graph can be achieved by a combination of inheritance link deletions and class deletion. The class can be referenced elsewhere in the schema through feature specification. Those features and the classes defining and inheriting them become shadow.

- **Add an Inheritance Link to a Class.**

  First, the environment checks that the new inheritance link does not induce a cycle in the inheritance graph. Then the features provided by the new superclass (either inherited or locally defined) are propagated along the new link and the same checks as for feature addition are performed.

- **Remove an Inheritance Link from a Class.**

  Removing a superclass from a class C can lead to inconsistencies in the schema due to the fact that a subclass relationship between a descendant of C (or C itself) and an ancestor

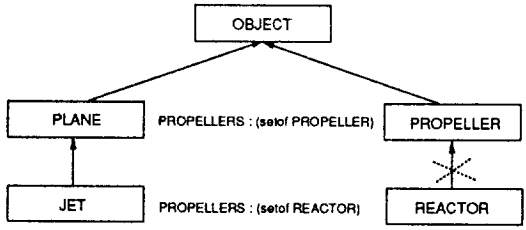

Figure 7: The "inheritance path break" problem

of C no longer holds. This kind of "inheritance path break" is illustrated in Figure 7. The inheritance link between the REACTOR class and the PROPELLER class is removed. This leads to a subtyping violation in the JET class since the redefinition of the "propellers" attribute is no longer valid.

Therefore, to accept the modification, everywhere in the inheritance graph that C or a subclass of C is used in a redefinition, the subclass check implied by the redefinition has to be performed again but without taking into account the removed inheritance link. If there still remains a path connecting the subclass of C to its presumed ancestor, there is no inconsistency. Moreover, the system also checks that the references to ancestors of C induced by "from" clauses in C and its subclasses are still valid. If the modification does not introduce inconsistencies, all the features inherited by the class and its subclasses via the removed link are deleted in them, except if they are still inherited through an alternative path. If the class is disconnected in the inheritance graph (i.e. with no superclasses), the OBJECT class is added as a default superclass.

4.4 Implementation

All information about the schema (classes, features, database variables) and its implementation (i.e. methods) is handled by a component of the $LISPO_2$ system, named the Schema Manager. The Schema Manager is implemented in the $LISPO_2$ language itself. The reason for this choice is twofold. First all information about the schema has to persist from one programming session to the other. Second, an object-oriented approach makes the implementation of the Schema Manager easier. It promotes a modular design (through information hiding) allowing the experimentation of different check algorithms. Moreover, inheritance allows code sharing and reusability. This is illustrated in Figure 8 which shows a portion of the "meta schema" used to represent class/feature definitions as objects. We can see the benefit of inheritance to gather the OPERATION class and the ATTRIBUTE class under the FEATURE class. This allows the sharing of all algorithms detecting the impact of feature modifications on the schema and methods. Moreover the late-binding mechanism, offering extensibility, allows us to easily add a new type constructor with its

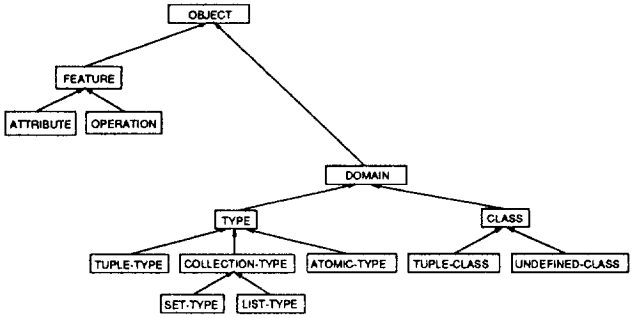

Figure 8: The meta schema

own subtyping rule. All we have to do is to define a class for this new type constructor, and to specify its local subtyping rule as an operation and its associated method. Thereafter, it will be immediately integrated into all of the schema modification framework.

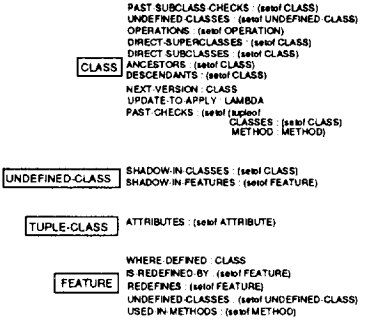

Figure 9: Some attributes defined by the meta schema

Figure 9 gives a subset of the definitions of the structures used to detect the inconsistencies in the schema induced by modifications. To handle fast name conflict detection, all features provided (i.e. defined or inherited) by a class are stored in the class (using the "operations" and "attributes" attributes defined respectively in the CLASS and TUPLE-CLASS classes). To detect name conflicts and distinguish locally defined features from inherited ones, the "where-defined" attribute (defined in the FEATURE class) stores the class where the feature is defined. To speed up redefinition checks (in the case of feature addition or feature replacement), each feature is linked to both its upward and downward redefinitions (through respectively the "redefines" and "is-redefined-by" attributes). This avoids walks in the inheritance graph to check for redefinition errors.
The inheritance graph is internally represented by the four "direct-superclasses", "direct-subclasses", "ancestors" and "descendants" attributes. The subclass check is then reduced to simply testing the membership of the presumed subclass in the descendants of the class.

The shadow mechanism is handled by several attributes and the UNDEFINED-CLASS class. Whenever an undefined class is used in a feature specification, an instance of the UNDEFINED-CLASS class is created. This instance acts as a stub recording all the checks that will have to be performed on the undefined class when it is defined. These checks are memorized by the operation implementing the subclass check when it is called on undefined classes. The shadow/deshadow detection is managed by the attributes "undefined-classes" defined in the FEATURE and CLASS classes. They record the undefined classes leading directly or indirectly (through a shadow class) to the shadow status of the feature/class. The pending cross-references are the "shadow-in-class" and "shadow-in-feature" attributes defined in the UNDEFINED-CLASS class. These attributes record respectively the classes and features which are shadow because of the undefined class. When a previously undefined class is introduced, the system sees if this new class is shadow or not. If the class is not shadow, it is removed from the cause of shadowness of its dependent features/classes. This removal can lead to their deshadowing. Otherwise, the set of undefined classes implying the shadow status of the new class replaces the class in the cause of shadowness of its dependent features/classes.

The detection of the inheritance path break problem (illustrated in Section 4.3) is handled by means of the "past-subclass-checks" attribute defined in the CLASS class. This attribute records all the superclasses which have been successfully tested as an ancestor of the class (due to feature redefinition for example). When a superclass is removed from a class, this attribute is scanned in the class and all its descendants, and the checks are performed again.

5 Repercussions of Schema Modifications on Methods

This section discusses the impacts of schema modifications on methods. Methods are coded in LISP extended with object manipulation expressions. Those expressions include creating objects, reading and writing attributes, and sending messages à la Smalltalk [Gold83]. Message sending involves the late-binding mechanism where the operation called depends on the class of the receiver. There also exists a "super" mechanism allowing the programmer to specify the starting class from which to look up the operation.

The $LISPO_2$ method compiler performs static type checking of the object expressions using type inferencing and user supplied type declarations (when needed). It catches any inconsistencies in the method with regard to the schema (e.g. detecting references to unknown attributes/operations). The type-checking algorithm allows each "variable" (i.e. formal argument of an operation, local variable or attribute) to be assigned an expression whose type is a subtype of the static (declared or inferred) type of the variable. Since the type-checking algorithm uses the schema at the moment

of the method compilation (i.e. the defined classes with their features and the subclass relationships represented by the inheritance graph), a schema modification can affect the type validity of previously compiled methods. Indeed, schema modifications may have two kinds of impact on a method. They can lead to new type errors or they can imply a change in the behavior of the method due to the late-binding mechanism.

When a schema modification is issued, the induced actions on methods performed by the programming environment can be of three kinds:

- Directly mark a method as invalid. In this case, the environment knows, without having to recompile the method, that the modification introduces new type errors in the method.
- Recompile a method to detect type errors. In this case, the environment does not have enough information to directly assert the invalidity of the method but it knows that the change may induce a type error. So recompilation is necessary and the method is marked as invalid if the type-checker discovers errors.
- Warn the developer since the modification may change the behavior of the method.

5.1 Persistent Root Modifications

Only removal of a persistent root affects methods. All methods referring to the removed root are directly marked as invalid.

5.2 Class Content Modifications

- **Add a Feature to a Class.**

  If there was no previously inherited feature with the same name in the class, i.e. if there is no upward redefinition, there is no impact on methods. Otherwise, all methods referring to the inherited feature are pointed out because their behavior may change due to the late-binding mechanism.

- **Remove a Feature from a Class.**

  If there is no inherited feature replacing the removed one, all methods referencing the

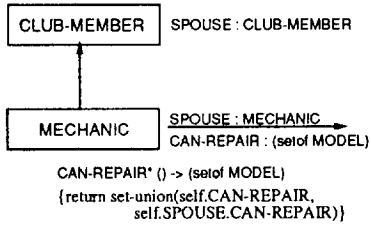

Figure 10: Type error induced by removing a feature

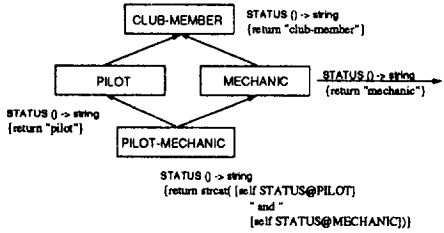

Figure 11: Change in behavior induced by removing a feature

removed feature are directly marked as invalid. Otherwise, there is a change analysis between the specification of the removed feature and the one of the newly inherited feature. If they are not the same, all methods referencing the removed feature are recompiled to discover new type errors. Figure 10 illustrates this case. In order to avoid disturbing the reader with syntactic details, the examples of methods use an abstract syntax where message sending is denoted by the "[]" brackets and attribute access by the dot notation. In this example, the programmer removes the "spouse" attribute defined in the MECHANIC class. The "spouse" attribute is then replaced in the MECHANIC class by the one defined in the CLUB-MEMBER class. However its domain is the CLUB-MEMBER class. This affects the "can-repair*" method computing all the plane models a mechanic can repair as the union of the plane models that the mechanic and his/her spouse can repair. Indeed, there is a loss of information (since the CLUB-MEMBER class does not provide the "can-repair" attribute), leading to a type error in the "can-repair*" method. In contrast, Figure 11 gives an example of a change behavior introduced by removing a feature. The "status" method is removed from the MECHANIC class. It is replaced by the one defined in the CLUB-MEMBER class which has the same signature but a different method. This leads to a changed behavior in the "status" method defined in the PILOT-MECHANIC class. This method calls the status operations provided by the MECHANIC and PILOT classes using the "super" mechanism (denoted by a @). So the result of this method is changed from "pilot and mechanic" to "pilot and club-member".

- **Change the Specification of a Feature in a Class.**

  There is a change analysis between the new and the old specification of the feature. If the new specification redefines the old one[1], all methods referencing the old feature are pointed out due to a potential changed behavior. Otherwise, these methods are recompiled in order to discover type errors.

[1]For an attribute, this means that the new domain is a subtype/subclass of the old one. For an operation, the redefinition rule holds between the new and the old signatures.

- **Rename a Feature in a Class.**

  If the renamed feature implies upward redefinitions, the methods referencing the newly redefined features are pointed out since they may suffer from a change in their behavior. Moreover, if an inherited feature with the old name appears in the class, a change analysis is performed between the specifications of the inherited and the renamed features as for a feature replacement.

5.3 Inheritance Graph Modifications

Creation of a class cannot induce type errors. The environment only points out the methods which reference features redefined in the new class, since their behavior may be affected. Deletion of a class leads to directly marking all methods referencing the class, either through one of its features or through its name (e.g. in a variable declaration), as invalid. Addition of an inheritance link can only induce potential redefinitions and thus a changed behavior in methods referencing the newly redefined features. In contrast, removal of an inheritance link from a class C can lead to an "inheritance path break" problem, i.e. C or one of its descendants is no longer a subclass of an ancestor of C. All methods which might be type inconsistent due to this modification are directly marked as invalid (as explained in Section 5.4). This is also the case for the methods referencing a feature no longer provided by the class (or its subclasses) due to the removal. Renaming a class has no impact on methods. This is due to the fact that methods, once compiled, refer directly to the class without using its name.

5.4 Implementation

When the compiler is invoked on a method, it computes a compiling context. This context records all properties of the schema which have been used to assert the type validity of the method. Such properties include the classes used, the features which must be provided by these classes, and the subclassing test performed. The cross-references implied by this context are represented by the "used-in-method" attributes (defined in the FEATURE and CLASS classes as shown in Figure 9) recording in which methods the feature/class is referenced. These attributes are used to find the methods to be directly marked as invalid (e.g. when a class is removed) or to be recompiled (e.g. when the specification of a feature is changed). Moreover, the "past-check" attribute (defined in the CLASS class) stores all subclass checks performed by the compiler to assert the type validity of the method. Therefore, when an inheritance link is removed from a class, all checks involving the class (or its descendants) with respect to one of its ancestors are selected from this attribute and are performed again.

```
(defun transform-mechanic ()
   (ifn (instance-of? old.spouse 'PILOT-MECHANIC)
        (setq new.spouse [old find-or-create-spouse])
        (setq new.spouse old.spouse)))
```

Figure 12: Transformation function example

6 Repercussions of Schema Modifications on Instances

6.1 Persistent Root Modifications

Persistent root modifications affect the set of references pointing to the instances and, thus, their deletion by the garbage collector. Adding a class extension makes persistent those instances of the class which were not referenced by the persistent roots, while removing a class extension (resp. a database variable) deletes objects if they are no longer referenced elsewhere in the database.

6.2 Class Content Modifications

Only modifications of attributes affect the structure of instances. Adding or renaming an attribute in a class leads to the logical addition of the attribute to all instances of the class and of the subclasses inheriting the attribute. Removing an attribute from a class implies its logical deletion from all instances of these classes if the attribute is not replaced by an inherited one.

Replacing the domain of an attribute (either directly by issuing a "replace" modification or indirectly by adding an attribute) does not affect instances if the new domain is a supertype/superclass of the old one. Otherwise, the object associated with the attribute in existing instances may not be of the new domain. For example, if we change the domain of the "spouse" attribute defined in the MECHANIC class from MECHANIC to PILOT, every instance having a mechanic as value of the "spouse" attribute violates the new class definition. In such cases, the default policy of the environment is to replace all values of the "spouse" attribute by a void reference. However, when the programmer issues the schema modification, he/she can specify a transformation function which will be applied to all affected instances. This function is in charge of computing a new value for the attributes which are affected by the schema modification. For example, the transformation function shown in Figure 12 does not change the value of the "spouse" attribute if it is an instance of the PILOT-MECHANIC class. Otherwise, it invokes the find-mechanic-spouse operation which retrieves or creates the mechanic spouse of the object. In a transformation function, two pseudo variables **old** and **new** are used. They represent respectively the old and new versions of the instance being transformed. The system ensures that a transformation function is performed only once for an object. This prevents infinite loops when cyclic objects are transformed.

6.3 Inheritance Graph Modifications

The impact of an inheritance link addition (resp. removal) on instances is reduced to a set of attribute additions (resp. removals). In contrast, removing a class raises the problem of what happens to the instances of the removed class. In $LISPO_2$, we delete them but this approach may lead to potential dangling references if those instances were referenced by objects of other classes. Therefore, all instances of the class are deleted and every object referring to a deleted instance through an attribute is updated with a void reference as the new value of the attribute (using a mechanism explained in Section 6.4).

6.4 Implementation

The restructuring of instances is based on a semi-lazy evolution policy. The modifications are immediately propagated to all instances which are in main memory. However, for the instances on disk, they are only performed when the instance is loaded in main memory (by an object fault mechanism). This policy results from a tradeoff between efficiency and interactivity. Immediate propagation of the modification on all instances would decrease the interactivity of the system if there is a great number of instances on disk. On the other hand, performing the propagation on demand would require checking, on every access to an object, if it has to be updated. This would decrease the performance of methods working on main memory objects. In order to perform the update check only once, a solution could be to flush all involved instances onto disk and to update them only at load-time (therefore the check has to be done once). Unfortunately, the current implementation of persistence in $LISPO_2$ uses a two address space model (i.e. an instance is identified by its RAM address in main memory and by a persistent identifier onto disk) as in PS-ALGOL [Atki81]. Therefore, the cost of the flushing step would be too high since we would have to convert main memory addresses into persistent identifiers when flushing instances on disk. This transformation would require the scan of the entire main memory.

The implementation of this semi-lazy policy requires two system facilities: the ability to enumerate all instances of a class in main memory and the capacity of storing instances of various versions of a class on disk. The first point is achieved by maintaining a class extension for each class (even if it has not been declared by the programmer). The extension in main memory chains all the direct instances of the class together and points to the extensions of its subclasses. The second point is handled by creating a version of the class after each modification. This is performed using the "next-version" and "update-to-apply" attributes defined in the CLASS class. The first attribute links the successive versions of a class. The second one stores the transformation to apply in order to make the instance evolve from the previous to the next version. Therefore, when an instance

is loaded from disk, the version of its class is compared to the version which is in main memory. If they differ, the chain of versions is followed and each update is applied. Class versions are objects and, as objects, they are reclaimed automatically by the garbage collector when there are no references (here instances since an instance holds a reference to its class) to them.
This class versioning mechanism is also used to incrementally avoid the dangling references problem after a class removal. When a modification deleting a class is issued, a new class version is created for each class referencing the removed class as the specification of an attribute. The associated transformation function is automatically generated by the system. Its role is to replace the value of the attribute by a void reference eliminating the dangling reference.

7 Related Work

Two major systems, namely ORION[Bane87b] and GemStone[Penn86], address the problem of impacts of schema modifications on the schema and its instances. Only ORION supports multiple inheritance. In this system, the means of solving name conflicts is a default rule based on the order of the superclasses. In particular, this rule is used to block the propagation of modifications (such as adding, renaming or replacing a feature) when it implies a name conflict in subclasses. Thus, a modification can be partially applied. In contrast, the LISPO_2 philosophy of propagation is "all or nothing". That is, the modification is applied everywhere the feature is inherited if it is possible, otherwise it is not applied at all. This respects the natural view of inheritance where a feature is shared by all the subclasses inheriting it.
In order to preserve the structure of instances, these two systems reduce the power of schema modifications. For example, in ORION, the domain of an attribute can only be generalized. In the same vein, GemStone does not allow the addition of an attribute, if there is already an inherited attribute with the same name. Moreover GemStone allows class deletion only if the class has no instances. In contrast, ORION deletes all instances of the class leading to the problem of dangling references. In LISPO_2, there is no reduction of the power of schema modifications due to their impacts on the instances. Moreover, our restructuring policy avoids the problem of dangling references after a class deletion, as explained in Section 6.4.
GemStone does not support the addition and deletion of inheritance links while these modifications are provided by ORION. However, the potential "inheritance path break" induced by deletion of a superclass is not mentioned in [Bane87b] and does not seem to be handled. Concerning class creation, none of these systems allows the programmer to work with a partially defined schema (i.e. the shadow mechanism).
Neither GemStone nor ORION addresses the issue of the impacts of schema modification on meth-

ods. In contrast, Encore[Skarr86] promotes an interesting approach relying on a class versioning and error handling mechanism to make the change transparent to methods. In this approach, the programmer defines a set of routines attached to a class version. These routines handle errors due to the mismatch between methods and the class version, such as accessing an unknown attribute or violating a domain constraint. In $LISPO_2$, the aim is, first of all, the automatic detection by the environment of the components of the schema and its implementation affected by a modification. This provides the programmer with a global view of the impacts of a change before it is performed. The approach in Encore can be seen as complementary. When the change has been confirmed, this approach can be applied to all methods where new type errors are introduced. However, the burden on the programmer of the class versioning and error handling mechanisms has to be taken into account.

Concerning the transformation of affected instances, our approach is very similar to the one in [Lern90]. The transformation function ensures the mapping between the old and new versions of instances. However, [Lern90] only deals with the structural aspect of schema modification and does not address the issue of method recompilation.

8 Conclusion and Future Directions

This paper has presented the schema modifications supported by the $LISPO_2$ programming environment. They allow the programmer to quickly develop a first version (even incomplete) of the application and they enable him/her to easily incorporate changes suggested by previous experiments. We have illustrated the inconsistencies which may be introduced by these modifications at the schema, method and object levels. Furthermore, we have described the data structures used to detect such inconsistencies.

From the first uses of our schema modifications, we have identified two main drawbacks. The first one concerns the lack of a methodological tool asserting the quality of a resulting schema. Indeed, the use of schema modifications leads to a valid schema whose structure may present anomalies such as redundancies in the inheritance graph. These anomalies can be accepted in the first steps of the design but must be eliminated in the final one. To remedy this, a tool acting like a "lint" program is needed to point out the weakness of the final schema in quality domains such as maintainability or reusability. The second drawback concerns the hard-wired semantics and coarse granularity of the schema modifications. For example, when the programmer removes a superclass link, he might want to keep in the class some attributes which will disappear, or to explicitly indicate a new superclass when the class become disconnected. Therefore, we are working on a "toolkit" approach to address the schema evolution issue. It consists of schema modifications of finer granularity and

of a means of combining them in sequences. The validity of the compound schema modification will be checked only at the end of the sequence based on analysis of the changes imposed on the schema. Such an approach will provide a much more open-ended framework for schema evolution allowing the customization and creation of new modifications.

References

[Agra89] R. Agrawal and N.H. Gehani, "ODE: The Language and the Data Model", *Proc. SIGMOD Conf.*, Portland, 1989.

[Atki81] M. Atkinson, "PS-ALGOL: an Algol with a Persistent Heap", *Sigplan Notices*, 17(7), July 1981.

[Banc88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard and F. Velez, "The Design and Implementation of O_2, an Object-Oriented Database System", *in Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.

[Bane87a] J. Banerjee, H.T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou and H.J. Kim, "Data Model Issues for Object Oriented Applications", *ACM Trans. Office Info. Syst.* 5(1), January 1987.

[Bane87b] J. Banerjee, W. Kim, H.J. Kim and H.F. Korth, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", *Proc. SIGMOD Conf.*, San Francisco, 1987.

[Barb90] G. Barbedette, "LISPO_2: A Persistent Object-Oriented Lisp", *Proc. 2nd EDBT Conf.*, Venice 1990

[Booc90] G. Booch, *Object-Oriented Design*, Benjamin/Cummings, 1990.

[Gold83] A. Goldberg and D. Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983.

[Cope84] G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proc. SIGMOD Conf.*, Boston 1984.

[Goss90] S. Gossain and B. Anderson, "An Iterative-Design Model for Reusable Object-Oriented Software", *Proc. OOPSLA Conf.*, Ottawa 1990.

[Lécl89a] C. Lécluse and P. Richard, "Modeling Complex Structures in Object-Oriented Databases", *Proc. PODS Conf.*, Philadelphia 1989.

[Lern90] B.S. Lerner and A.N. Habermann, "Beyond Schema Evolution to Database Reorganization", *Proc. OOPSLA Conf.*, Ottawa 1990.

[Meye88] B. Meyer, *Object Oriented Software Construction*, Prentice Hall, 1988.

[Penn86] D. J. Penney and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS", *Proc. 1st OOPSLA Conf.*, Portland 1986.

[Skarr86] A.H. Skarra and S.B. Zdonik, "The Management of Changing Types in an Object-Oriented Database", *Proc. 1st OOPSLA Conf.*, Portland 1986.

[Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.