Supporting Exceptions to Behavioral Schema Consistency to Ease Schema Evolution in OODBMS

Eric Amiel — INRIA Rocquencourt, BP 105, F-78153, Le Chesnay Cedex, Amiel@rodin.inria.fr

Marie-Jo Bellosta — Lamsade, Université Paris Dauphine, F-75775 Paris Cedex 16, bellosta@lamsade.dauphine.fr

Eric Dujardin — INRIA Rocquencourt, BP 105, F-78153, Le Chesnay Cedex, Dujardin@rodin.inria.fr

Eric Simon — INRIA Rocquencourt, BP 105, F-78153, Le Chesnay Cedex, Simon@rodin.inria.fr

Abstract

Object-oriented databases enforce behavioral schema consistency rules to guarantee type safety, i.e., that no run-time type error can occur. When the schema must evolve, some schema updates may violate these rules. In order to maintain complete behavioral schema consistency, traditional solutions require significant changes to the types, the type hierarchy and the code of existing methods. Such operations are very expensive in a database context. To ease schema evolution, we propose to support exceptions to the behavioral consistency rules without sacrificing type safety for all that. The basic idea is to detect unsafe statements at compile-time and check them at run-time. The run-time check is performed by a specific clause that is automatically inserted around unsafe statements. This check clause warns the programmer of the safety problem and lets him provide exception-handling code. Schema updates can therefore be performed with only minor changes to the code of methods.

Keywords : Object-oriented databases, schema evolution, type safety, covariance, contravariance.

Proceedings of the 20th VLDB Conference Santiago, Chile, 1994

1 Introduction

An object-oriented database schema contains the description of the types[1], type hierarchy, and methods used by all application programs. To ensure static type checking, object-oriented systems typically enforce that a schema satisfies three ***behavioral consistency*** rules. These rules are ***sufficient*** conditions that guarantee that no type error can occur during the execution of a method code. The ***substitutability*** rule says that if a type T_1 is a subtype of a type T_2 then whenever an instance of T_2 is expected in a variable assignment or a function invocation, it must be allowed to pass an instance of T_1. The ***covariance*** and ***contravariance*** rules impose constraints when a method is redefined for more specialized types. The covariance rule says that the return type be also specialized. The contravariance rule says that the types of arguments that are not used for late binding must be more general. If a database schema satisfies these rules, it is said to be ***behaviorally consistent***.

A typical situation in object-oriented databases is that the schema must evolve in order to accommodate evolutions of the application programs. As argued in [Bor88], this is particularly important in databases "where it is in general impossible or undesirable to anticipate all possible states of the world during schema design". The problem is that some schema updates may violate the behavioral consistency rules. For example, consider a database schema that contains a type ***Patient*** having an attribute ***doctor*** of type ***Physician***. Suppose that we define a new type, called ***Alcoholic***, as a subtype of ***Patient*** and such that the attribute ***doctor*** inherited from ***Patient*** is redefined to be of type ***Psychologist***. Since a ***Psychologist*** is (usually) not a ***Physician***, the method that retrieves the ***doctor*** attribute value of an alcoholic violates the covariance rule and the method that updates the ***doctor*** attribute value of an alcoholic violates the

[1] We intentionally avoid to talk about classes, which are viewed as types in some systems and as type extensions in others.

contravariance rule.

There are also specific situations that are part of the (real-life) application that constitute violations of the behavioral consistency rules. For instance, in an hospital database, one may say that ambulatory patients are exactly like patients (i.e., *Ambulatory_patient* is a subtype of *Patient*) except that they have no hospital ward. This leads to violate the substitutability rule because the method that retrieves a *ward* attribute value is not applicable to an instance of *Ambulatory_patient*.

Existing systems have two attitudes with respect to this problem. One is to encourage the programmer to follow the rules but not actually force him to do so (e.g., C++, or O_2 for the contravariance rule). Inconsistent schemas are allowed and it is the programmer's responsibility to control what the program does and avoid run-time type errors. The second attitude is to prevent the user from violating the rules. In this case there are several well-known solutions that lead to either change the type hierarchy and introduce "fake" types, or break the type hierarchy and loose the advantages of polymorphism. These solutions may require significant changes to the code of methods. Following [Bor88], we believe that both attitudes are clearly not satisfactory since they result in either unsafe code or substantial and artificial revisions to the schema.

The starting point of our research is that *exceptions* to the behavioral consistency rules should be supported to ease schema evolution and modelling. However, they should be controlled to avoid type safety problems. In this paper, we propose to have a tool that processes every method source code and (i) determines whether a statement is unsafe, i.e., may result in a run-time type error, (ii) automatically inserts a "check" clause around every unsafe statement in the source code, and (iii) let the user provide exception-handling code. The check clause is merely an if-then-else statement where the if-part performs a safety run-time check, the then-part contains the original statement, and the else-part contains the exception-handling code[2]. The insertion of check clauses warns the user about possible run-time type errors. The safety condition in the if-part of the "check" clause is expressed intensionally, thereby avoiding to reformulate the condition when the schema changes. Our tool can also automatically generate some default exception-handling code. However, if the programmer provides his/her own exception-handling code then it has to be inspected by our tool.

[2]We do not focus on the issue of designing specific language primitives for handling exceptions that can be harmoniously integrated with existing OO programming languages.

Our proposed approach facilitates schema evolution by supporting exceptions while guaranteeing that no run-time type error will occur. We focus on the motivations for such an approach and the type checking of statements in the presence of exceptions to behavioral consistency. Our results apply to object-oriented databases that support run-time method selection using either a single method's argument (mono-methods) or all method's arguments (multi-methods) as in recent systems like CLOS [BDG+88], Polyglot [ADL91], and Cecil [Cha92].

The paper is organized as follows. Section 2 introduces preliminary definitions about single and multi-targetted methods, and defines the notion of behaviorally consistent schema. Section 3 gives an overview of the problem whereas Section 4 sketches the proposed solution. Section 5 introduces the material necessary to present our type system. Section 6 describes the type checking process allowing to distinguish between safe and unsafe statements. Section 7 relates our work with existing work, and Section 8 concludes the paper.

2 Behavioral Schema Consistency

In this section, we introduce our notations for the types and methods of a schema, mostly defined in [ADL91]. Then, we define the behavioral schema consistency rules and their exceptions.

2.1 Notations

We assume the existence of a partial ordering between types, called ***subtyping*** ordering, denoted by $\preceq$. Given two types T_1 and T_2, if $T_1 \preceq T_2$, we say that T_1 is a subtype of T_2 and T_2 is a supertype of T_1. To each generic function m corresponds a set of methods $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$, where T_k^i is the type of the i^{th} formal argument, and where R_k is the type of the result. We call the list of arguments $(T_k^1, \ldots, T_k^n)$ of method m_k the ***signature*** of m_k. An invocation of a generic function m is denoted $m(T_1, ..., T_n)$, where $(T_1, \ldots, T_n)$ is the signature of the invocation, and the T_i's represent the types of the expressions passed as arguments. We shall use uppercase letters to denote type names, and lowercase letters to denote type instances, generic functions, methods and method invocations.

In traditional object-oriented systems, functions have a specially designated argument, the ***target***, whose run-time type is used to select the method to execute (method resolution). Multi-methods, first introduced in CommonLoops [BKK+86] and CLOS [BDG+88], provide a generalization of single-targetted methods by making all arguments targets. Multi-methods are now a key feature of several systems such as Polyglot [ADL91], Kea [MHH91], and Cecil [Cha92]. Henceforth, we consider that methods are targetted on either one or all arguments. For the sake of uniformity, we shall assume that the p first arguments of a function (where $p = 1$ or $p = n$) are the target arguments. In the examples, we underline the target arguments in the signatures.

$$\begin{tikzcd} Person & equal_1(\underline{Person}, Person) \\ Employee \arrow[u] & equal_2(\underline{Employee}, Employee) \end{tikzcd}$$

Figure 1: A simple schema

Example 2.1 Consider the type hierarchy of Figure 1, and suppose we wish to define a generic function *equal* on persons and employees. Since equality is defined differently for persons and employees, two methods $equal(\underline{Person}, Person)$ and $equal(\underline{Employee}, Employee)$ are needed to implement the generic function and we respectively denote them $equal_1$ and $equal_2$. Their signatures, given on Figure 1, show that these methods have a single target argument. On invocation *equal(Person, Employee)*, the run-time method dispatcher will select method $equal_1$ based on the first target argument. □

Given a generic function invocation, the selection of the corresponding method follows a two-step process : first, based on the types of the target arguments, a set of applicable methods is found and second, a precedence relationship between applicable methods is used to select what is called the *Most Specific Applicable* method (MSA). Intuitively, a precedence relationship determines which applicable method most closely matches a function invocation. Given a function invocation m and a particular method precedence ordering noted $<$, if m_i and m_j are two applicable methods and m_i is more specific than m_j, noted $m_i < m_j$, then m_i is the closest match for the invocation.

In the rest of this paper, we assume that for any function invocation $m(T_1, \dots, T_n)$, if there is an applicable method, then there always exists a *Most Specific Applicable* (henceforth, MSA) method and this method is unique. We call this the Unique Most Specific Applicable (UMSA) property. However, we insist that our results do not depend on the means by which the UMSA property is enforced.

Usually, types are represented using different data structures such as set, tuple and list. We assume that for each type, there is a set of (representation) operations that enable to manipulate (i.e., access and update) the structure of instances of that type. For example, if T is a type represented as a tuple : $[a_1 : T_1, \dots, a_n : T_n]$, where the a_i's denote attribute names, then the generic functions a_i and set_a_i, respectively access and update the a_i attribute value of an instance of T.

2.2 Behavioral Consistency Rules

Object-oriented typing theory defines three consistency rules to guarantee that no type error can occur during the execution of a method code. A database schema is said to be ***behaviorally consistent*** (in the following, ***consistent***) iff every method satisfies the consistency rules. The first two rules impose constraints on types returned by methods and the types of methods's arguments. The third rule relaxes the condition of type equality on substitution operations (variable assignment or parameter passing) to take into account the subtyping relationship. It is the basis of inclusion polymorphism [CW85]. The three rules are :

- **Covariance rule** : Given two methods $m_i(T_i^1, \dots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, \dots, T_j^n) \rightarrow R_j$, where $m_i < m_j$, then $R_i \preceq R_j$.
- **Contravariance rule** : Given two methods $m_i(T_i^1, \dots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, \dots, T_j^n) \rightarrow R_j$, where $m_i < m_j$, then $\forall k > p$, $T_j^k \preceq T_i^k$ (p is the number of target arguments).
- **Substitutability rule** : An instance of T_1 can be substituted to an instance of T_2 if and only if $T_1 \preceq T_2$ (***substitutability condition***).

Note that imposing the covariance and the substitutability rules on the methods accessing the structure of types amounts to enforce the rules of structural subtyping defined by [CW85]. For instance, according to these rules, a tuple-structured type T_1 may be a subtype of T_2 if T_1 has all the attributes of T_2, and if the types of common attributes in T_1 are subtypes of those in T_2. As a consequence, representation functions available on T_2 instances are also available on T_1 instances. Covariance also entails the ***domain compatibility invariant*** defined by [BKKK87]. This invariant states that the domain of an attribute that is redefined in a subtype can only be specialized. The contravariance rule was originally developed for subtyping of functions [Car84], and has been extended to subtyping on partially targetted methods in [McK92, Dan90]. The substitutability rule is also called ***strict*** or ***full inheritance*** invariant in [BKKK87].

3 Problem Overview

In this section, we first define exceptions to schema consistency and show how they arise with some schema updates. Then, we summarize the safety problems induced by exceptions. Finally, we present solutions recommended by object-oriented design methods to avoid schema inconsistencies.

3.1 Exceptions to Consistency and Schema Updates

We define an *exception* as the violation of one of the three consistency rules. The non-respect of the covariance rule yields ***return-exceptions*** while the non-respect of the contravariance rule yields ***argument-exceptions***. Violations of the substitutability rule yields two kinds of exceptions. The first one is when a signature is disallowed for a generic function, although the substitutability condition for parameter passing is satisfied. The second one is when the substitutability condition is violated during assignment or parameter passing. These exceptions are respectively called ***disallowed signature*** and ***illegal substitution***.

In the following, we only consider return-exceptions, argument-exceptions, and disallowed signatures as possible exceptions to the consistency rules. Indeed, illegal substitutions have more far-reaching consequences on static type checking than the three other kinds of exceptions.

3.1.1 Return-exceptions

Method m_i is a *return-exception* to method m_j if $m_i < m_j$ and the return type of m_i is not a subtype of the return type of m_j.

Imposing covariance on the results ensures that whatever method is selected at run-time, its result is a subtype of the type expected by the context of the invocation. An interesting case of return-exception is the violation of the domain compatibility invariant.

$$\begin{tikzcd}
 & Doctor & & Patient \\
Physician \arrow[ur] & & Psychologist \arrow[ul] & Alcoholic \arrow[u, "\preceq"']
\end{tikzcd}$$

$doctor_1(\underline{Patient}) \rightarrow Physician$
$doctor_2(\underline{Alcoholic}) \rightarrow Psychologist$
$set\_doctor_1(\underline{Patient}, Physician)$
$set\_doctor_2(\underline{Alcoholic}, Psychologist)$
$hospital_1(\underline{Physician}) \rightarrow Hospital$

Figure 2: Doctor and Patient hierarchy

Example 3.1 Suppose that *Patient* is a type having an attribute *doctor* of type *Physician*. Suppose we want to add a new type *Alcoholic* to the schema as a subtype of *Patient*, where attribute *doctor* is of type *Psychologist*. The updated schema is shown on Figure 2. We have an exception because *Psychologist* is usually not a subtype of *Physician*. Thus, the method $doctor_2$ is a return-exception to method $doctor_1$. This exception can cause type errors as shown below. Consider the method that takes a set of patients and refunds their expenses to the hospital they were treated in using method $refund(\underline{Hospital}, \underline{Dollar})$:

```
refunding(patients :  set[Patient])
{ for p in patients do
    refund(hospital(doctor(p)),bill(p));
  end do }
```

As Psychologists are not affiliated to an hospital unlike Physicians, the invocation ***hospital(doctor(myPatient))*** causes an error if ***myPatient*** refers to an alcoholic at run-time as there is no applicable method for invocation ***hospital(Psychologist)***. □

3.1.2 Argument-exceptions

Method m_i is an *argument-exception* to method m_j if $m_i < m_j$ and there exists a non-target argument T_i^k of m_i which is not a supertype of T_j^k.

Argument-exceptions only occur in systems with single-targetted functions where run-time method selection does not check that the non-target arguments of an invocation are subtypes of the non-target formal arguments of the selected method. This may result in illegal substitutions when the actuals are assigned to the formals. However, the possibility to specialize any argument of a method is clearly needed in practice and for this reason, most object oriented systems do not actually enforce the contravariance constraint (see [CCPLZ93], [Mey92], [CM92], [$O_2$92]).

Example 3.2 In the schema of Figure 1 where *Employee* is a subtype of *Person*, suppose we have a method $equal_1(\underline{Person}, Person)$ targetted on the first argument and the schema is updated by adding a new method $equal_2(\underline{Employee}, Employee)$. Then, invocation $equal(myPerson_1, myPerson_2)$ leads to the selection of $equal_2$ if the target argument, $myPerson_1$, refers to an employee at run-time. But if the type of $myPerson_2$ refers to a person, an illegal substitution occurs between the formal argument of type *Employee* and $myPerson_2$. Then, in the body of $equal_2$, applying on this argument a function that is only defined for *Employee* (e.g. to access an attribute specific to *Employee*) causes a run-time error as there is no applicable method. □

3.1.3 Disallowed Signatures

Signature s is a *disallowed signature* of m if invoking m on s is forbidden, although there exists an MSA method for $m(s)$.

Example 3.2 has shown that some signatures should be disallowed because they imply illegal substitutions between non-target actual and formal arguments. We refer to these signatures as ***implicitly*** disallowed signatures as they can be inferred from argument-exceptions. However, some disallowed signatures cannot be inferred and must be explicitly given by the user as part of the semantics of its application.

This is the case with inapplicable attributes (see e.g., [Bor88]). We call these signatures *explicitly* disallowed signatures.

$$\begin{tikzcd} & Person & & & Beverage & \\ Professor \arrow[ur, "\preceq"] & & Student \arrow[ul, "\preceq"'] & Juice \arrow[ur, "\preceq"] & & Alcohol \arrow[ul, "\preceq"'] \end{tikzcd}$$

Figure 3: Disallowed signature

Example 3.3 Consider the schema of Figure 3 and suppose we update the schema by adding a function *purchase* with a single associated method $purchase(\underline{Person},\underline{Beverage})$. This method should naturally not be applicable to signature (*Student*, *Alcohol*), which is disallowed. All other signatures are allowed. □

$$\begin{tikzcd} Person & & List[Person] \\ Student \arrow[u] \arrow[rr, phantom, "\Rightarrow"] & & List[Student] \arrow[u] \end{tikzcd}$$

$insert_1(\underline{List[Person]}, \underline{Person}) \rightarrow List[Person]$

$insert_2(\underline{List[Student]}, \underline{Student}) \rightarrow List[Student]$

Figure 4: Type constructor

Example 3.4 Consider the schema of Figure 4. Signature (*List*[*Student*], *Person*) violates the composition integrity constraint on constructed types : for a constructed type *A*[*T*], its components objects are required to be subtypes of *T*. Thus, this signature must be disallowed for *insert*. Note that it would be implicitly disallowed if the first argument was the only target. □

3.2 Inconsistent Schemas and Type Safety

A program is type safe if, during the execution of every statement, no error can occur due to the absence of an MSA for a method invocation. The purpose of static type checking is to verify at compile-time that a program is type safe. To this end, for each statement of a method code, the declared types are used to check that (i) every invocation has an MSA, and (ii) no illegal substitution may occur. If the two above conditions are satisfied, a statement is correct. Otherwise, it is incorrect and there is a type error.

The central problem introduced by exceptions to consistency is that a correct statement may be unsafe, i.e., yield a type error at run-time. Thus, in presence of exceptions to consistency, type checking must further partition correct statements into *safe* and *unsafe* statements.

$$\begin{tikzcd} \text{Absence of MSA} & \text{Illegal Substitution} \arrow[l] & \text{Invocation with an Explicitly Disallowed Signature} \\ \text{Return-Exception} \arrow[u] \arrow[ur] & \text{Argument-Exception} \arrow[u] & \text{Explicitly Disallowed Signature} \arrow[u] \end{tikzcd}$$

Figure 5: Exceptions and Type Errors

Figure 5 summarizes the relationships between the three different kinds of exceptions to consistency (bottom of figure) and the three kinds of type errors at run-time (top of figure) : an arrow from x to y means that an exception of kind x may lead to a type error of kind y at run-time.

3.3 Existing Solutions to Avoid Inconsistent Schemas

Object-oriented design offers several solutions to the problems of inconsistency set by some schema updates. They consist in modifying the type hierarchy and the code of methods or introducing new methods. These solutions avoid return-exceptions and explicitly disallowed signatures, but not argument-exceptions. However, they involve important modifications of the type hierarchy or the code of methods. In a database context, this can be expensive since changes to the types must be propagated to the persistent instances. Most importantly, the burden of implementing these solutions is left to the programmer. We examine four of these solutions on Example 3.1.

The first solution eludes the problem by renouncing to make *Alcoholic* a subtype of *Patient*. Thus, the advantages of polymorphism are lost : alcoholics and patients must be stored in different sets and they must be handled separately, by different methods, despite their similarities.

The second solution retains the advantages of polymorphism for the methods that use only the similarities between *Alcoholic* and *Patient*. This solution involves a new intermediate type to represent the common part, in our case *Patient* without attribute *doctor*. This can be achieved in two ways, illustrated in Figure 6 : (i) modify *Patient* by removing attribute *doctor* and create a subtype *Patient_treated_by_Physician*, or (ii) create *Patient0* as a supertype of *Patient*, to represent patient without attribute *doctor*. In both cases, *Alcoholic* is made a subtype of the intermediate type. In methods that do not use the difference between alcoholics and regular patients and that do not call methods using this difference, patients and alcoholics can be manipulated as being of the intermediate type. Such methods can

$$\begin{tikzcd}
& \textit{Patient} & \\
\textit{Alcoholic} \arrow[ur] & & \textit{Patient\_treated\_by\_Physician} \arrow[ul]
\end{tikzcd}$$

$doctor_1(\underline{Patient\_treated\_by\_Physician}) \rightarrow Physician$
$doctor_2(\underline{Alcoholic}) \rightarrow Psychologist$

$$\begin{tikzcd}
& \textit{Patient0} & \\
\textit{Patient} \arrow[ur] & & \textit{Alcoholic} \arrow[ul]
\end{tikzcd}$$

$doctor_1(\underline{Patient}) \rightarrow Physician$
$doctor_2(\underline{Alcoholic}) \rightarrow Psychologist$

Figure 6: Intermediate Supertype Creation

be termed as *non-critical*, whereas in our previous example, *refunding* is critical.

The first problem with this solution is the multiplication of artificial intermediate types, like *Patient0*, which is combinatorial in nature (see [Bor88]) as they represent objects with a subset of the attributes of *Patient*. The second problem is that retaining polymorphism through the use of an intermediate type only works for non-critical methods. In critical methods, the intermediate type cannot be used. In our previous example, every method that calls *refunding* is critical and cannot pass a heterogeneous set containing both regular patients and alcoholics. This is a major disadvantage in a database context, where applications are collection-oriented. In this case, solution (ii) is preferable because it only requires to modify methods but not existing instances.

The third solution consists in re-conciliating physicians and psychologists by declaring a method *hospital* on *Doctor*. This method is defined as simply returning a NULL reference to indicate that doctors who are psychologists are affiliated to no hospital. This way, invocation *hospital*(*doctor*(*p*)) is not an error even if *p* refers to an alcoholic at run-time. The problem with this solution is the definition of artificial methods, like *hospital*(<u>*Doctor*</u>), which seems to indicate that a function is available on a certain type while it is actually not. Moreover, it is the responsibility of the programmer to know that *hospital* invoked with a doctor may return a NULL reference and that the result of the function must be tested. In our example, *refunding* must be rewritten as :

```
{ for p in patients do
      if hospital(doctor(p)) <> NULL
          refund(hospital(doctor(p)),
                 bill(p));
  end do }
```

A last solution consists in defining two intermediate methods *foo*(<u>*Patient*</u>) and *foo*(<u>*Alcoholics*</u>). The first encapsulates the original statement refunding the hospital, the second defines what must be done in the case of an alcoholic. Method *refunding* is then rewritten to call *foo* on patients :

```
refunding(patients :  set[Patient])
{ for p in patients do
      foo(p);
  end do }

foo(p:Patient)
{ refund(hospital(doctor(p)),bill(p));}

  foo(p:Alcoholic)
{/* handles the case of alcoholics */}
```

The problem with this solution is the multiplication of artificial switching methods.

4 The Proposed Solution

We propose to accept unsafe statements due to exceptions while guaranteeing that no type error can occur at run-time. The idea is to embed every statement identified as unsafe at compile-time into a *check statement* of the following form :

```
CHECK <condition>
 <unsafe statement>
ELSE
 <exception-handling code>
END
```

The condition part checks at run-time that the unsafe statement is correct and if it is, the statement is executed. Otherwise, an exception-handling code is executed. Check statements enable to warn the user about the possibility of run-time failure, let the user provide exception handling code, and perform dynamic type checking of the unsafe statement.

Throughout this paper, we consider statements that are either function invocations or variable assignments as shown below by the pseudo-EBNF grammar :

statement	::=	*assignment* | *invocation*
assignment	::=	*variable* ← *expression*
invocation	::=	*function_name*(*expression**)
expression	::=	*variable* | *constant* | *invocation*

For an unsafe invocation $m(e_1, \ldots, e_n)$, the condition part of the check statement is :

m IS CORRECT ON $(e_1, \ldots, e_n)$

For an unsafe assignment $v \leftarrow e$, the condition part of the check statement is :

```
e MAY BE ASSIGNED TO v
```

Example 4.1 In Example 3.1, invocation *hospital*(*doctor*(*myPatient*)) was unsafe because *myPatient* may contain an alcoholic. Thus, this statement will be surrounded by the following check :

```
CHECK hospital IS CORRECT
            ON (doctor(myPatient))
   hospital(doctor(myPatient))
ELSE
   Exception Handling Code
END    □
```

Using check statements minimizes the modifications of the schema that are needed to accommodate exceptions to consistency. There is no need to create new types, add new artificial methods, or renounce polymorphism by not declaring a type as a subtype of another one. Moreover, the changes to the code of existing methods is minor and can be automatized by a compiler, unlike solutions where the programmer must test the result of methods that may return NULL values.

Example 4.2 Suppose that a new type of physician, *FamilyPractitioner*, is introduced, on which *hospital* is not applicable (i.e., an explicitly disallowed signature). As our correction test is intensional, the check does not need to be reformulated. On the contrary, if explicit reference to the exception on alcoholic were made, as in [Bor88], the check would have to be changed from "CHECK *myPatient* IS NOT *Alcoholic*" to "CHECK *myPatient* IS NOT *Alcoholic* AND *doctor*(*myPatient*) IS NOT *FamilyPractitioner*". □

This example demonstrates the advantage of an intensional expression of checks over an extensional one. By not mentioning types, adding or removing a method or a disallowed signature does not require reformulating the check statement. The correctness condition is evaluated using the state of the schema at run-time (we implicitly assume that the schema can be queried at run-time).

We assume that verifying the correctness of a statement at run-time has no side-effects. As dynamic type checking involves evaluating arguments, which may be invocations, we assume that only functions without side-effect are used as invocation's arguments of unsafe statements. Otherwise, temporary variables must be used.

In summary, for every statement, the proposed type checking process works as follows :

1. Determine whether the statement is incorrect, unsafe or safe.
2. If the statement is incorrect, report the type error.
3. If the statement is unsafe, generate the appropriate check statements.
4. Prompt the user for exception-handling code.
5. Type check the statements of the exception-handling code.

In the first step, determining if a statement is correct relies on the types known at compile time, while determining if it is safe relies on the potential types at run time. In the third step, the generation of the check statement must consider that several subexpressions of a statement may be unsafe. In such cases, check statements must be nested. The main problem with nested checks is to avoid unnecessary checks. When unsafe subexpressions share some variables or some subexpressions, checks may become redundant. The general idea is to allow the type checker to infer the possible run-time types of sub-expressions along a chain of nested checks (equivalent to a chain of conditionals). The fourth step is deferred so that the user gives, at the same time, the exception-handling code for all unsafe statements. In the fifth step, the types inferred along the checks are used to type-check the exception-handling code in place of the types known at compile time. Because of space limitations, we only describe the first step of this process.

5 Basic Definitions

In this section, we introduce the notions of method applicability, exact type and cover of a signature.

Total Match and Target Match. Let $m_k(T_k^1, \ldots, T_k^n)$ and $m(T_1, \ldots, T_n)$ be respectively a method and a function invocation for a generic function m. Then, m_k is said to be a *total match* for the invocation iff $\forall i \in \{1, \ldots, n\}$, $T_i \preceq T_k^i$, and m_k is said to be a *target match* for the invocation iff $\forall i \in \{1, \ldots, p\}$, $T_i \preceq T_k^i$ (p is the number of target arguments).

By extension, we talk about a method as being a total or target match for a signature. Note that in multi-targetted systems, the two notions merge, i.e., every target is a total match.

Method Applicability. A method $m_k(T_k^1, \ldots, T_k^n)$ is *applicable* to a function invocation $m(T_1, \ldots, T_n)$ if and only if m_k is a target match for the invocation.

Consider again Figure 1 and suppose that *equal* is invoked with *equal*(*Employee*, *Person*). Both methods $equal_1$ and $equal_2$ are applicable because they are both target match to this invocation. However,

$equal_1(\underline{Person}, Person)$ is a total match for the invocation and $equal_2(\underline{Employee}, Employee)$ is not a total match.

In the following, we use a function MSA which, given an invocation $m(T_1, \ldots, T_n)$, returns the most specific applicable method m_k for this invocation if any, and a specific method "$m_\perp$" otherwise. The method $m_\perp$ uses a specific "impossible" type, noted $T_\perp$, as the type of its arguments and result. $T_\perp$ is in strict supertype relation with all other types, i.e., $\forall T,\ T \prec T_\perp$. This special method is defined for every generic function. MSA is used at run-time as the method dispatcher.

We now introduce the notion of *exact type* of an expression. The type of a constant c declared of type T is *exactly* T and not any type $T' \preceq T$. Similarly, the object resulting from an explicit "new" creation instruction is exactly the type given as argument to "new". Thus, a variable that gets assigned the result of a "new" instruction is also of an exact type. Exact typing applies to expressions that appear as actual arguments of invocations or as right-hand side of assignments.

Exact Typing. At compile-time, an expression e is said to be of an *exact type* T, denoted $e : \overline{T}$, iff any object referenced by e at run-time is of type T and not of any type T' such that $T' \preceq T$.

Note that, by default, any expression e is of *free type* T, denoted $e : T$, i.e., e may yield at run-time an object of any type $T' \preceq T$. We shall use letter τ to indifferently refer to $\overline{T}$ and T when typing an expression.

Signature of Expressions. The signature of an n-tuple of expressions $(e_1 : \tau_1, \ldots, e_n : \tau_n)$ is the n-tuple $(\tau_1, \ldots, \tau_n)$. The signature of a method $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$ is the signature of its formal arguments, i.e., $(T_k^1, \ldots, T_k^n)$. The signature of an invocation $m(e_1, \ldots, e_n)$ with $e_1 : \tau_1, \ldots, e_n : \tau_n$ is the signature of its actual arguments, i.e., $(\tau_1, \ldots, \tau_n)$. Abusively, we shall call *signature* any n-tuple of free or exact types $(\tau_1, \ldots, \tau_n)$, and omit their associated expressions.

Cover of a Signature. Let s be a signature $(\tau_1, \ldots, \tau_n)$. The *cover* of s, denoted by $cover(s)$ is defined as :

$$cover(s) = \{(U_1, \ldots, U_n) \mid \forall i \in \{1, \ldots, n\} \begin{cases} U_i \preceq T_i & \text{if } \tau_i = T_i \ (\tau_i \text{ is free}) \\ U_i = T_i & \text{if } \tau_i = \overline{T}_i \ (\tau_i \text{ is exact}) \end{cases}\}$$

By extension, we also define the cover of a method m_i as the cover of its signature. Note that $cover(m_i)$ is the set of signatures for which m_i is a total match.

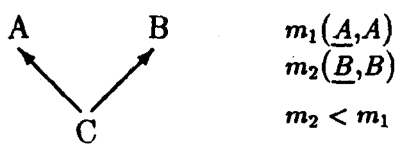

Figure 7: Example Schema

Example 5.1 Using the type hierarchy in Figure 7, we have :

$$cover(A, \overline{A}) = \{(A, A), (C, A)\}$$
$$cover(m_1) = cover(A, A) = \{(A, A), (A, C), (C, A), (C, C)\}$$
$$cover(m_2) = cover(B, B) = \{(B, B), (B, C), (C, B), (C, C)\} \qquad \square$$

6 Type Checking with Exceptions

In this section, we consider the type checking of statements in the presence of exceptions to consistency. To specify type checking we use a generic function called *check*. It has four methods to respectively handle constants, variables, assignments of the form $t \leftarrow e_1$ and invocations of the form $m(e_1, \ldots, e_n)$, where each e_i is an expression. The result of each *check* method is either *incorrect*, *safe* or *unsafe*. As trivial cases, the result for constants and variables is *safe*.

The last two methods (i.e., for assignments and invocations) proceed in two steps. The first step evaluates the safety of the statement using the types of the expressions e_i known at compile time, also called the *static types*. If the statement is found to be safe, then its safety is further evaluated in the second step. This step uses the potential types, at run-time, of the expressions e_i composing the statement. These types are called the *dynamic types*.

The distinction between the static and dynamic types is required in the presence of return-exceptions. When covariance of the result types is respected, the type of an invocation known at compile-time is the unique most general type that the invocation may have at run-time. This is not true when a method is allowed to return a type that is not a subtype of the types returned by more general methods. Going back to Example 3.1, the invocation *doctor(myPatient)* has *Physician* for its static return type. However, due to the return-exception *doctor*$_2$, its possible types at run-time are not only the subtypes of its static type *Physician*, but also the subtypes of *Psychologist*. Thus, its dynamic types are *cover(Physician)* $\cup$ *cover(Psychologist)*.

This section is organized as follows. First we detail the type checking algorithms for assignments and invocations. They are based on the type checking of *reduced* statements, i.e. statements where the expressions e_i of the input statements are replaced by their static or dynamic types. We then specify the type-checking of a reduced statement. Finally, we define the static and dynamic types of expressions.

6.1 Static Type Checking of Assignments

To type check an assignment $v \leftarrow e$, the first step replaces v and e by their static types which are computed by function ***static***. The resulting reduced statement is then checked using function ***check$_R$***. If it is incorrect or unsafe, i.e., not safe, then $v \leftarrow e$ is respectively incorrect or unsafe. Otherwise, its safety must be further probed using the type at run-time of the right-hand side, e. An assignment can be unsafe for two reasons : (i) e is not safe, or (ii) e may return, at run-time, a type that is not a subtype of the type of v. The set of most general types that e may evaluate to at run-time is computed using function ***dynamics***.

```
check(v ← e) /* check for assignments */
input: an assignment v ← e
output: incorrect, safe or unsafe
Step 1: /*Safety w.r.t static types : replace v and e
          by their static type using static */
   reducedAssignment := ( static(v) ← static(e) ) ;
   result:= check_R( reducedAssignment );
   if result is not safe
      return result ;
Step 2: /*Safety w.r.t run-time types*/
   if check(e) is not safe
      return unsafe ;
   /* Replace the right-hand side by each of its most
      general types at run-time using dynamics */
   for each T ∈ dynamics(e) do
      reducedAssignment := ( static(v) ← T ) ;
      if check_R( reducedAssignment) is not safe
         return unsafe ;
   end do ;
   return safe ;
   end check
```

6.2 Static Type Checking of Invocations

To type check an invocation $m(e_1, \dots, e_n)$, the first step replaces its arguments which are computed by their static types. The resulting reduced invocation is then checked using function ***check$_R$***. If it is incorrect or unsafe, i.e., not safe, then $m(e_1, \dots, e_n)$ is respectively incorrect or unsafe. Otherwise the invocation is staticallyꞁcorrect and its safety must be further evaluated in the second step. At this step, the invocation may be unsafe for two reasons : (i) there exists an unsafe argument e_i or (ii) for some signature at run-time, the invocation is not safe. Otherwise, the invocation is safe. Function ***signatures*** computes the set of most general signatures that may appear as arguments of a method invocation at run-time.

```
check(m(e_1, ... , e_n)) /* check for invocations */
input: an invocation m(e_1, ... , e_n)
output: incorrect, safe or unsafe
Step 1: /*Safety with respect to static types : replace the
          arguments by their static type using static */
   reducedInvocation := ( m(static(e_1), ... , static(e_n))) ;
   result:= check_R(reducedInvocation) ;
   if result is not safe
      return result ;
Step 2: /*Safety with respect to run-time types */
   for each argument e_i do
      if check(e_i) is not safe
         return unsafe ;
   end do ;
/* Using signatures, replace the arguments by each of the
   most general signatures at run-time */
   for each s ∈ signatures(m(e_1, ... , e_n)) do
      reducedInvocation := m(s) ;
      if check_R (reducedInvocation) is not safe
         return unsafe ;
   end do ;
   return safe ;
   end check
```

6.3 Type Checking Reduced Statements

A reduced assignment is an expression of the form $T_1 \leftarrow \tau_2$ while a reduced invocation is an expression of the form $m(s) = m(\tau_1, \dots, \tau_n)$. The type checking of reduced assignments is defined as follows.

$$check_R(T_1 \leftarrow \tau_2) = \begin{cases} \text{safe} & \text{if } \tau_2 \preceq T_1 \\ \text{unsafe} & \text{if } (T_1) \in cover(\tau_2) \\ \text{incorrect} & \text{otherwise} \end{cases}$$

$$check(m(s)) = \text{incorrect if} \begin{cases} MSA(m(s)) = m_\perp \text{ or} \\ MSA(m(s)) \text{ is not a total match for } m(s)\text{, or} \\ s \text{ is explicitly disallowed for } m \end{cases}$$

Note that we allow assignments where the static type of the right-hand side is a supertype of the type of the left-hand side variable. Such unsafe assignments are similar to the reverse assignment of Eiffel [Mey92] or the dynamic downward cast of C++ [Laj93].

The safety of a reduced invocation is defined as follows :

$$check(m(s)) = \begin{cases} \text{safe} & \text{iff } \forall s' \in cover(s)\ check(m(s')) \neq \text{ incorrect} \\ \text{unsafe} & \text{otherwise} \end{cases}$$

6.4 Static and Dynamic Types of an Expression

The static type of an expression can now be defined as shown on Figure 8.

Example 6.1: Consider again the types and methods of Figure 2 of Section 3. Let *refund(<u>Hospital</u>, <u>Dollar</u>)* be the method used in Example 3.1 to refund

Constant c	$static(c) = \overline{T}$
Variable v	$static(v) = T$
Reduced Invocation $m(s)$	$static(m(s)) = \begin{cases} T_{\perp} \text{ if } check(m(s)) = \text{incorrect} \\ \text{return type of } m_k \\ \quad = \text{MSA(m(s)) otherwise} \end{cases}$
Invocation $m(e_1, \ldots, e_n)$	$static(m(e_1, \ldots, e_n)) = static(m(static(e_1), \ldots, static(e_n)))$

Figure 8: Static Type of Expressions

the expenses of patients to hospitals. The first step in the type-checking of invocation $refund(hospital(doctor(p)), amount)$, where p is a variable of type *Patient* and *amount* a variable of type *Dollar*, consists of computing the static types of the arguments $hospital(doctor(p))$ and *amount* as follows :

$$\begin{aligned} static(hospital(doctor(p))) &= \\ static(hospital(static(doctor(p)))) &= \\ static(hospital(static(doctor(static(p))))) &= \\ static(hospital(static(doctor(Patient)))) &= \\ static(hospital(Physician)) &= Hospital \\ \text{and} \qquad static(amount) &= Dollar \end{aligned}$$

As $check(refund(Hospital, Dollar)) \neq$ incorrect, invocation $refund(hospital(doctor(p)), amount)$ is correct. □

We now formally define the dynamic types of an expression as shown on Figure 9. The set of dynamic types of a reduced invocation contains only the highest types that can be returned by the invocation at run-time. By highest, we mean types that are not subtypes of any other type in the set (we use operator $max_{\preceq}$ to obtain the highest types in a set of types).

Constant c	$dynamics(c) = \{\overline{T}\}$
Variable v	$dynamics(v) = \{T\}$
Reduced Invocation $m(s)$	$dynamics(m(s)) = max_{\preceq}\{R_i \mid m_i \in RTC(m(s))\}$
Invocation $m(e_1, \ldots, e_n)$	$dynamics(m(e_1, \ldots, e_n)) = max_{\preceq}(\bigcup_{s \in signatures(m(e_1, \ldots, e_n))} dynamics(m(s)))$

Figure 9: Dynamic Types of an Expression

The definition of the dynamic types of a reduced invocation $m(s)$ relies on the notion of *run-time correct* methods. They represent the methods that can be selected at run-time for *correct* invocations covered by $m(s)$.

Run-Time Correct Methods. Let $m(s)$ be a reduced invocation.

$$RTC(m(s)) = \{MSA(m(s')),\ s' \in cover(s) \mid check(m(s')) \neq \text{incorrect}\}$$

The definition of the dynamic types of an invocation $m(e_1, \ldots, e_n)$ relies on the set of signatures that may appear at run-time as arguments of the invocation. As usual, this set contains only the highest signatures, all the signatures in their cover being implicitly included. This set is denoted $signatures(m(e_1, \ldots, e_n))$ and consists of the cross product of the dynamic types of the invocation's arguments :

Signatures of an Invocation. The set of highest signatures that may appear at run-time for an invocation is :

$$signatures(m(e_1, \ldots, e_n)) = \prod_{i=1}^{n} dynamics(e_i)$$

Example 6.2 : The second step in the type-checking of invocation $refund(hospital(doctor(p)), amount)$ starts by type checking $hospital(doctor(p))$ and *amount*. First, $hospital(static(doctor(p)))$=$hospital(Physician)$ is neither incorrect or unsafe. Thus the safety of $hospital(doctor(p))$ must be checked. To this end, the algorithm determines the signatures of $hospital(doctor(p))$.

$$\begin{aligned} &signatures(hospital(doctor(p))) \\ &\qquad = \{(T) \mid T \in dynamics(doctor(p))\} \\ &\qquad = \{(Physician), (Psychologist)\} \end{aligned}$$

One of the signatures of $hospital(doctor(p))$, namely Psychologist, makes the invocation incorrect as there is no MSA. Thus $hospital(doctor(p))$ is unsafe. So finally, as one of its arguments is unsafe, $refund(hospital(doctor(p)), amount)$ is unsafe.

7 Related Work

The problems due to maintaining consistency rules have been recognized by many researchers, each focusing on a particular rule, but to our knowledge, considering these problems in a single framework has never been proposed.

[Coo89, McK92] forbid argument-exceptions. Hence, subtyping between generic collections (list of *Person* and list of *Student*) and attribute type redefinition are also disallowed.

Esse [CPLZ91, CCPLZ93] and Eiffel [Mey92] use data flow analysis to detect unsafe invocations due to argument-exceptions : the set of types to which a variable may refer (called *type set* in [CPLZ91, CCPLZ93] and *dynamic class set* in [Mey92]), is maintained during type checking and evaluated after every statement. Using this "type-flow" technique, a slightly larger class of programs are statically determined to be safe as exact types may be used to replace constant objects or variables that have just been assigned a newly created object. Although this approach provides more accurate type checking, two problems remain. First, statements that cannot be proved to be safe are rejected (pessimistic option). Second, this approach is less applicable to a database context where applications use collections. Indeed, a variable iterating over a collection of T may refer to objects of any subtype of T with no way of knowing the exact subset of types present in the collection. Our approach can be used as a complement to "type flow" techniques, taking over when they have failed to prove the safety of a statement.

Using a special construct called *reverse assignment*, Eiffel [Mey92] allows a certain kind of illegal substitution : the assignment of an expression with static type T_1 to a variable of type T_2, although T_1 is a supertype of T_2. The assignment is checked at run-time to ensure that the dynamic type of the expression is actually T_2 or a subtype of T_2. Otherwise, a NULL reference is assigned to the variable. It is the responsibility of the programmer to check that the variable is not NULL after the reverse assignment. A similar construct, the *dynamic cast* [Laj93], is being incorporated into C++ to check at run-time the correctness of a *down-ward cast* (assertion by the programmer that an object of static type T_1 is actually of type T_2 with T_1 supertype of T_2).

[CM92] uses bounded type quantification, restricting the application of subtyping to enforce the composition integrity constraint on constructed types. Bounded universal quantification allows substitutability only when passing parameters to a function. All other assignments must involve objects of the same type. Bounded existential quantification extends substitutability to assignments in the called function. In all cases, bounded quantification requires the exact types of actual parameters to be known statically. It is this knowledge that allows static type checking of covariant code. In particular, this prevents passing bounded parameters to another function.

In the works on *method schemas* [AKW90, Wal91], no consistency rules are imposed on the schema and the return type of user-defined methods is not specified. Consistency is defined as type safety, i.e., absence of run-time type errors. Proving type safety involves simulating the execution of methods from a typing point of view. This is shown to be impossible in the general case,i.e., with multi-targetted methods and recursion. Covariant updates are shown to maintain consistency.

Our approach is very similar to [Bor88] in that it aims at detecting unsafety at compile-time, using dynamic type checking when necessary and allowing the user to write exception handling code. [Bor88] addresses the problem of inapplicable attributes and return-exceptions due to attribute domain redefinition. The notion of *excuses* serves to distinguish between desired exceptions and errors. A type system that supports these excuses is formally defined in [Bor89], along with an efficient algorithm to statically detect unsafe statements. Check clauses are provided by the user. He/she formulates the correction condition in an extensional way, testing the run-time type of expressions. The type system verifies that the correction condition implies the safety of the checked statement and of the exception-handling code. We extend this work in two directions. First, we address the problem of exceptions on single- and multi-targetted methods. Second, we provide an intensional formulation of the correction condition, allowing this condition to remain invariant when the type hierarchy is modified and/or new exceptions are introduced.

8 Conclusion

In this paper, we proposed to facilitate schema evolution in object-oriented databases by supporting exceptions to behavioral schema consistency while at the same time guaranteeing type safety. After presenting the three consistency rules of covaraiance, contravariance and substitutability, we defined a typology of exceptions. We gave examples of schema updates that naturally yield exceptions to the consistency rules, and we showed that existing solutions that seek for preserving schema consistency lead to expensive restructurations of the type hierarchy and method codes. We then proposed a new type checking process whereby exceptions to consistency can be safely tolerated. To guarantee type safety, every statement is first analyzed to determine if it is correct or not and then further analyzed to determine if it is safe or not. Then, every unsafe statement is surrounded by a check clause. This clause is merely an if-then-else statement where the if-part performs a run-time type checking, the then-part contains the original statement, and the else-part contains some exception-handling code (user-defined or system-generated).

Unlike traditional solutions offered by object-oriented design, our approach enables to handle schema updates that do not preserve schema consistency without creating artificial types and methods or modifying the type hierarchy. Schema updates can only yield the additions of check clauses in the code of existing methods. Another advantage of our solution is that conditions in the check are specified intensionally, thereby avoiding to reformulate them when the type hierarchy is modified or when exceptions are introduced or removed. We believe our approach provides a useful complement to existing sophisticated techniques for static type checking. Indeed, our proposed system relieves these techniques when they fail to prove the safety of a statement. Finally, we are not aware of any other work in the field of object-oriented systems and languages that considered exceptions to schema consistency in the general framework of mono and multi-targetted functions.

All the steps of the proposed type checking process have now been specified (see [ABDS94]). Future work involves providing the user with means to express explicitly disallowed signatures, and developing efficient algorithms to implement our type checking. Finally, an environment to help programming with exceptions is being designed. Such an environment adresses important issues such as providing the user with explanations about why some statements are unsafe and assistance in writing exception-handling code.

Acknowledgments: We would like to thank François Bancilhon and Guy Ferran for their interest in this work. Special thanks go to Catriel Beeri, Françoise Fabret, Claude Delobel and Patrick Valduriez for their insightful comments on an earlier version of this paper.

References

[ABDS94] E. Amiel, M.-J. Bellosta, E. Dujardin, and E. Simon. Supporting exceptions to behavioral schema consistency to ease schema evolution in OODBMS. To appear as INRIA Research Report, 1994.

[ADL91] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static type checking of multi-methods. In *Proc. OOPSLA*, 1991.

[AKW90] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas. In *Proc. ACM PODS*, 1990.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *SIGPLAN Notices*, Sept. 1988.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Proc. OOPSLA*, 1986.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. ACM SIGMOD*, 1987.

[Bor88] A. Borgida. Modeling class hierarchies with contradictions. In *Proc. ACM SIGMOD*, 1988.

[Bor89] A. Borgida. Type systems for querying class hierarchies with non-strict inheritance. In *Proc. ACM PODS*, 1989.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Proc. Int. Symp. on Semantics of Data Types, LNCS 173*, 1984.

[CCPLZ93] F. Cattaneo, A. Coen-Porisini, L. Lavazza, and R. Zicari. Overview and progress report of the ESSE project : Supporting object-oriented database schema analysis and evolution. In *Proc. TOOLS*, 1993.

[Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *Proc. ECOOP*, 1992.

[CM92] R.C.H. Connor and R. Morrison. Subtyping without tears. In *Proc. Australian Computer Science Conference*, 1992.

[Coo89] W. Cook. A proposal to make Eiffel typesafe. In *Proc. ECOOP*, 1989.

[CPLZ91] A. Coen-Porisini, L. Lavazza, and R. Zicari. Updating the schema of an object-oriented database. *IEEE Data Engineering Bulletin*, 14:33–37, 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[Dan90] S. Danforth. Multi-targetted virtual functions for OODB. In *Proc. Journées Bases de Données Avancées*, 1990.

[Laj93] J. Lajoie. The new language extensions. *C++ Report*, July-August 1993.

[McK92] R. McKenzie. *An Algebraic Model of Class, Inheritance, and Message Passing*. PhD thesis, Computer Science Dept., University of Texas at Austin, 1992.

[Mey92] B. Meyer. *EIFFEL : The Language*. Prentice Hall Intl., 1992.

[MHH91] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In *Proc. ECOOP*, 1991.

[$O_2$92] O_2 Technology. *The O_2 User's Manual*, 1992.

[Wal91] E. Waller. Schema updates and consistency. In *Proc. Intl. Conf. on Deductive and Object-Oriented Databases*, 1991.