Type-Safe Relaxing of Schema Consistency Rules for Flexible Modelling in OODBMS

Eric Amiel, Marie-Jo Bellosta, Eric Dujardin, Eric Simon

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Type-Safe Relaxing of Schema Consistency Rules for Flexible Modelling in OODBMS.

Eric Amiel, Marie-Jo Bellosta, Eric Dujardin, Eric Simon

N ˚ 2638

Août 1995

PROGRAMME 1

Rapport de recherche

ISSN 0249-6399

Type-Safe Relaxing of Schema Consistency Rules for Flexible Modelling in OODBMS.

Eric Amiel [*], Marie-Jo Bellosta [**], Eric Dujardin [*], Eric Simon [*]

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Rodin

Rapport de recherche n˚ 2638 — Août 1995 — 34 pages

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 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. The basic idea is to detect unsafe statements in a method code 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.

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

(Résumé : tsvp)

To Appear in the VLDB journal special issue, August 1995

[*] {amiel}{dujardin}{simon}@rodin.inria.fr
[**] Lamsade, Université Paris Dauphine, 75775 Paris Cedex 16, bellosta@lamsade.dauphine.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Relachement des règles de cohérence de schéma avec typage sûr pour une modélisation souple dans les SGBDOO

Résumé : Les bases de données à objets imposent aux schémas des règles de cohérence qui garantissent la sûreté de typage, c'est-à-dire qu'aucune erreur de type ne peut survenir à l'exécution. Lorsque le schéma doit évoluer, certaines mises à jour peuvent violer ces règles. Pour conserver la cohérence comportementale du schéma, les solutions traditionnelles nécessitent des modifications significatives des types, de leur hiérarchie ainsi que du code des méthodes pré-existantes. Ces opérations sont très coûteuses pour une base de donnée. Afin de faciliter l'évolution de schéma, nous proposons de supporter des exceptions à ces règles de cohérence, sans sacrifier la sûreté de typage pour autant. L'idée essentielle est de détecter les instructions non sûres à la compilation, et de les vérifier à l'exécution. La vérification est effectuée grace à une opération particulière qui entoure automatiquement toute instruction non sûre. Cette opération avertit le programmeur du problème de sûreté, et lui permet de fournir le code de traitement de l'exception. Les mises à jour du schéma peuvent ainsi être effectuées, avec seulement des modifications mineures du code des méthodes.

Mots-clé : Bases de données orientées-objet, évolution de schéma, typage sùr, covariance, contravariance.

1 Introduction

An object-oriented database schema contains the description of the types[1], type hierarchy, and methods used by all application programs. Types and method interfaces allow to model the complex objects coming from conceptual design, while method code and type representation define the implementation of objects. As a consequence, object-oriented databases must meet requirements arising from both a conceptual data modelling and a programming perspective.

From a programming point of view, it is highly desirable to guarantee type safety, for instance in order to protect the database against data corruption caused by type errors. To ensure type safety, 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_2 is a subtype of a type T_1 then whenever an instance of T_1 is expected in a variable assignment or a function invocation, it must be allowed to pass an instance of T_2. The *covariance* and *contravariance* rules impose constraints when a method is redefined for more specialized types. The covariance rule says that the return type must also be 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*.

However, from a database modelling perspective, the schema must evolve in order to accommodate evolutions of the real world. 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 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. Both attitudes

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

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 checked at run-time to avoid type errors. In this report, we propose to process every method source code and (i) determine whether a statement is unsafe, i.e., may result in a run-time type error, (ii) automatically insert 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.

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 report is organized as follows. Section 2 introduces preliminary definitions about single and multi-targeted methods, and defines the notion of 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 describes how this process can be optimized. Section 8 establishes the relationships between the notions of consistency and safety. Section 9 relates our work with existing work, and Section 10 concludes the report.

2 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 consistency rules and how they impact on structural consistency through encapsulation. Note that our notion of consistency is only concerned with typing, and not with semantics. It does not include issues such as integrity constraints (e.g., as in [FM94]) or business rules.

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 $T_2 \preceq T_1$, we say that T_2 is a subtype of T_1 and T_1 is a supertype of T_2. As in [ZM89, Bru93, DS92], subtyping is a declared relationship between types, which is decoupled from implementation decisions, and used solely to reflect operational similarities between different types.

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 ith formal argument, and where R_k is the type of the result. We call the list of

[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.

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$^+$86] and CLOS [BDG$^+$88], provide a generalization of single-targeted methods by making all arguments targets. Multi-methods are now a key feature of several systems such as Polyglot [DCL$^+$93], Kea [MHH91], Cecil [Cha92], Dylan [App94] and SQL3 [Mel94]. Henceforth, we consider that methods are targeted 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.

Figure 1: A simple schema

$$\begin{tikzcd} Person \\ Student \arrow[u] \end{tikzcd}$$

$equal_1(\underline{Person}, Person)$
$equal_2(\underline{Student}, Student)$

Example 2.1 : Consider the type hierarchy of Figure 1, and suppose we wish to define a generic function $equal$ on people and students. Since equality is defined differently for people and students, two methods $equal(\underline{Person}, Person)$ and $equal(\underline{Student}, Student)$ 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 $equal\ (Person, Student)$, the run-time method dispatcher will select method $equal_1$ based on the first target argument. $\square$

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 signature $s = (T_1, \ldots, T_n)$ and a function invocation $m(s)$, if m_i and m_j are applicable to $m(s)$ and, according to a particular method precedence ordering, m_i is more specific than m_j for s, noted $m_i <_s m_j$, then m_i is a closer match for the invocation. When the method precedence ordering does not depend on signatures, i.e. $\forall s\ m_i <_s m_j$, we just write $m_i < m_j$.

In the rest of this report, we assume that for any function invocation $m(T_1, \ldots, T_n)$, 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. [ADL91] examines different possible method precedence orderings, and focuses on global type precedence and inheritance order precedence, which enforce the UMSA property in case of multiple inheritance and multiple targets. However, we insist that our results do not depend on the means by which the UMSA property is enforced.

Types can be represented using different data structures such as set, tuple and list. We assume that the system enforces the *encapsulation* of the representation of types. Each type has a set of built-in *representation* operations that enable to manipulate (i.e., access and update) the state of instances of that type. For our purpose, we consider a subset of the operations defined in the ODMG object model [Cat94]. *Representation* methods perform built-in operations on each type. The table in Figure 2 summarizes the signatures of their representation methods. Moreover, it is possible to iterate over the elements of a collection, i.e. a set or a list, by using a $foreach$ statement.

Figure 2: Signatures of Representation Methods

Representation of type T	Update	Access
$T = tuple(\ldots, a_i : T_i, \ldots)$	$set\_a_i(\underline{T}, T_i)$	$a_i(\underline{T}) \rightarrow T_i$
$T = set(T_1)$	$insert\_element(\underline{T}, T_1)$ $remove\_element(\underline{T}, T_1)$	$empty?(\underline{T}) \rightarrow Boolean$
$T = list(T_1)$	$insert\_element(\underline{T}, T_1)$ $remove\_element(\underline{T}, T_1)$	$empty?(\underline{T}) \rightarrow Boolean$ $retrieve\_element\_at(\underline{T}, Integer) \rightarrow T_1$

Example 2.2 : As shown in Figure 3, both a Person and a Student have several income resources used to compute taxes. The total amount of their financial resources is also divided between a bank account and a life insurance. Additionally, $Students$ have a cardID. The following invocation allows to insert a new resource r in the resources list of a person p : $insert\_element(resources(p), r)$. □

Figure 3: Representation of Types

$Person : tuple(name : String,$
$onBankAccount : Float,$
$inLifeInsurance : Float,$
$resources : List(Resources))$

$Student : tuple(name : String,$
$onBankAccount : Float,$
$inLifeInsurance : Float,$
$resources : List(Resources),$
$cardID : String)$

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. The first two rules impose constraints on the types returned by methods and the types of methods 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. The three rules are :

- **Covariance rule** : Given two methods $m_i(T_i^1, \ldots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, \ldots, T_j^n) \rightarrow R_j$, where, for some signature s, $m_i <_s m_j$, then $R_i \preceq R_j$.
- **Contravariance rule** : Given two single-targeted methods ($p = 1$), $m_i(T_i^1, \ldots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, \ldots, T_j^n) \rightarrow R_j$, where, for some signature s, $m_i <_s m_j$, then $\forall k \geq 2$, $T_j^k \preceq T_i^k$.
- **Substitutability rule** : Given two types T_1 and T_2, an instance of T_2 can be substituted to an instance of T_1 if and only if $T_2 \preceq T_1$ (*substitutability condition*).

The covariance rule is called consistency in [ADL91]. The contravariance rule was originally developed for subtyping of functions [Car84], and has been extended to subtyping on partially targeted methods in [McK92, Dan90]. The substitutability rule is the basis of inclusion polymorphism [CW85].

2.3 Structural Consistency Rules

As shown in [KM94], the behavioral consistency rules on representation methods imply structural consistency rules on the representation because of encapsulation. These rules state that the representation of the supertypes must be included in the representation of their subtypes, and disallow the redefinition of attribute types of tuples and element types of collections. Let T_i and T_j be two types, such that $T_i \preceq T_j$, we have :

- **Tuple Subtyping Rule**: If the representation of T_j is $tuple(a_j^1 : T_j^1, \ldots, a_j^{n_j} : T_j^{n_j})$, then the representation of T_i is $tuple(a_i^1 : T_i^1, \ldots, a_i^{n_i} : T_i^{n_i})$, with $\{a_i^1, \ldots, a_i^{n_i}\} \subseteq \{a_j^1, \ldots, a_j^{n_j}\}$, and for all $k_j \leq n_j$, all $k_i \leq n_i$, $a_j^{k_j} = a_i^{k_i} \Rightarrow T_j^{k_j} = T_i^{k_i}$.
- **Set Subtyping Rule**: If the representation of T_j is $set(T_j^1)$, then the representation of T_i is $set(T_i^1)$, with $T_i^1 = T_j^1$.
- **List Subtyping Rule**: If the representation of T_j is $list(T_j^1)$, then the representation of T_i is $list(T_i^1)$, with $T_i^1 = T_j^1$.

These rules restrict the rules of structural subtyping defined by [CW85], that also appear in [BKKK87]. The rules in [BKKK87] state that a tuple-structured type T_2 is a subtype of T_1 iff T_2 has all the attributes of T_1, and if the types of common attributes in T_2 are subtypes of those in T_1. Thus, representation methods available on T_1 instances are also available on T_2 instances. Howerver, as noted in [KM94], the update operations do not respect the contravariance rule. [ZM89] generalizes this problem to the redefinition of method parameters with subtypes, called *specialization via constraints*. They show that specialization via constraints leads to run-time type errors that cannot be handled by type checking at compile-time.

3 Problem Overview

In this section, we first define exceptions to behavioral consistency and give several examples of each kind of exceptions. Next, we relate the violations of structural consistency rules to behavioral exceptions. We then summarize the type errors possibly induced by these exceptions. Finally, we present solutions recommended by object-oriented design methods to avoid exceptions to consistency.

3.1 Exceptions to Behavioral Consistency

We define a *behavioral exception* as the violation of one of the three behavioral 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 behavioral 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 iff $m_i <_s m_j$ for some signature s, and the return type of m_i is not a subtype of the return type of m_j.

Imposing covariance on the result ensures that whatever method is selected at run-time, its result is a subtype of the type expected by the context of the invocation.

Example 3.1 : Consider the schema in Figure 4, which respects the structural consistency rules. Consider the generic function $retrieve$ that searches a document database according to the profile of the library user and his topic of interest. A person receives a survey, while a student is presented the course book relevant to his level. Thus, the method $retrieve_2$ is a return-exception to $retrieve_1$, capable of yielding type errors. For instance, suppose that a generic function $build\_abstract$ uses pattern matching to extract the abstract of a paper and course books have no abstract. The statement $build\_abstract(retrieve(aPerson,$"$database\ systems$"$))$ leads to a run-time error if $aPerson$ refers to a student at run-time, as there is no applicable $build\_abstract$ method. □

Figure 4: Document Hierarchy

$$\begin{tikzcd}
Person & & Document & \\
Student \arrow[u] & CourseBook \arrow[ur] & & Paper \arrow[ul] \\
& & & Survey \arrow[u]
\end{tikzcd}$$

$$retrieve_1(\underline{Person},\underline{String}) \rightarrow Survey$$
$$retrieve_2(\underline{Student},\underline{String}) \rightarrow CourseBook$$
$$build\_abstract(\underline{Paper}) \rightarrow Abstract$$

Return-exceptions can also cause illegal substitutions which can then lead to run-time type errors.

Example 3.2 : Consider the following assignment of a variable $mySurvey$ of type $Survey$: $mySurvey \leftarrow retrieve(myPerson,$ "$database\ systems''$)$. If $myPerson$ refers to a student at run-time, a course book is assigned to $mySurvey$, which constitutes an illegal substitution. The invocation $build\_abstract(\ mySurvey)$ has no applicable method, thereby causing a run-time type error. □

3.1.2 Argument-exceptions

Method m_i is an *argument-exception* to method m_j iff $m_i <_s m_j$ for some signature s, 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-targeted 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 actual arguments are assigned to the formal arguments. 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 3.3 : Consider the schema of Figure 1 where $Student$ is a subtype of $Person$. The invocation $equal(myPerson_1, myPerson_2)$ leads to the selection of $equal_2$ if the target argument, $myPerson_1$, refers to a student at run-time. But if the type of $myPerson_2$ refers to a person, an illegal substitution occurs between the formal argument of type $Student$ and $myPerson_2$. Then, in the body of $equal_2$, applying on this argument a function that is only defined for $Student$ (e.g. to access the $cardID$ attribute) causes a run-time error as there is no applicable method. Note that the representation of types $Student$ and $Person$ conform to the structural subtyping rules. □

3.1.3 Disallowed Signatures

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

Example 3.3 shows 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 the application. We call these signatures *explicitly* disallowed signatures. Following [Bor88], they are defined as excuses on the generic function : $excuse\ m\ on\ s_1,...,s_x$.

Figure 5: Disallowed signature

$$\begin{tikzcd}
 & Person & & & Resource \\
Professor \arrow[ur] & & Student \arrow[ul] & & Salary \arrow[u] \\
 & Teaching\text{-}Assistant \arrow[ul] \arrow[ur] & & & Grant \arrow[u]
\end{tikzcd}$$

Example 3.4 : Consider the schema of Figure 5 where $Professor$ and $Teaching\text{-}Assistant$ have the same structure as $Person$ with an additional attribute Dept. Moreover, $Teaching\text{-}Assistant$ also has a $cardId$ attribute, like $Student$. Suppose we update the schema by adding a function $allocate$ that updates the $resources$ attribute, and manages the financial resources by distributing money between the bank account and the life insurance, depending on a complex criterion. This function has two methods $allocate_1(\underline{Person}, Resource)$ and $allocate_2(\underline{Professor}, Salary)$. A professor receives a salary and some grants are allocated for his research projects. A student can also receive a salary and/or a grant. A teaching-assistant can only receive a salary. Thus, the method $allocate_2$ is not applicable to signature ($Teaching\text{-}Assistant, Grant$), which is disallowed. Finally, the specialization of the second argument induces two implicitly disallowed signatures ($Professor$,$Resource$) and ($Teaching\text{-}Assistant$,$Resource$). All other signatures are allowed. □

3.2 Exceptions to Structural Consistency

Because of encapsulation, exceptions to structural consistency entail exceptions to the behavioral consistency rules. There are two kinds of exceptions to structural consistency: *data structure mismatch* and *component type redefinition*. A data structure mismatch arises in two cases: (i) when different data structures are used to build the representation, and (ii) in the case of *inapplicable attributes*, i.e., attributes of the supertype that do not appear in the subtype (see e.g. [Bor88]). A component type redefinition arises when a subtype has the same data structure as its supertype. This redefinition focuses on the types of tuples' attributes and collections' elements.

3.2.1 Data Structure Mismatch

In the case of a data structure mismatch, some or all of the representation methods of the supertype cannot be applied to objects of the subtypes. This corresponds to explicitly disallowed signatures. Figure 6 summarizes these disallowed signatures in the case of different

data structures between a type T_1 and its subtype T_2. Finally, disallowed signatures due to an inapplicable attribute $a_i : T_1^i$ of a type T_1 with respect to its subtype T_2 are (T_2) and (T_2, T_1^i) respectively for a_i and set_a_i.

Figure 6: Explicitly Disallowed Signatures for Tuples, Sets and Lists

Representation of supertype T_1	built-in methods	explicitly disallowed signatures
$T_1 = tuple(\ldots, a_i : T_1^i, \ldots)$	set_a_i	(T_2, T_1^i)
	a_i	(T_2)
$T_1 = set(T)$	$insert\_element$	(T_2, T)
	$remove\_element$	(T_2, T)
	$empty?$	(T_2)
$T_1 = list(T)$	$insert\_element$	(T_2, T)
	$remove\_element$	(T_2, T)
	$empty?$	(T_2)
	$retrieve\_element\_at$	$(T_2, Integer)$

Example 3.5 : Consider the schema of Figure 7, borrowed from [DS92]. A data structure mismatch occurs between $Polygon$ and $Square$, because a square is obviously a kind of polygon, but the data structure of these types differ. Hence the representation methods of $Polygon$ are not applicable to squares. □

Figure 7: Data Structure Mismatch

$$\begin{tikzcd} Point & Polygon \\ & Square \arrow[u] \end{tikzcd}$$

$$\begin{aligned} &Point = tuple(x : Integer, y : Integer) \\ &Polygon = list(Point) \\ &Square = tuple(upper\_left\_corner : Point, \\ &\qquad side\_length : Integer) \end{aligned}$$

Example 3.6 : Suppose we are given a schema where $Ambulatory\_Patient \preceq Patient$ and we want to update the schema by adding an attribute $ward$ for $Patient$. This attribute is irrelevant to subtype $Ambulatory\_Patient$. Thus, accessing or updating the ward of an $Ambulatory\_Patient$ should not be allowed. Then, $(Ambulatory\_Patient)$ and $(Ambulatory\_Patient, Ward)$ are respectively disallowed signatures for methods $ward(Patient) \rightarrow Ward$ and $set\_ward(Patient, Ward) \rightarrow Ward$. □

3.2.2 Component Type Redefinition

As shown in [KM94, Coo89, CMM91, DS92], a component type redefinition between a type T_1 and its subtype T_2 leads to one of the following exceptions :

- a return-exception of the access methods if the type appearing in T_2 is not a subtype of the corresponding type in T_1.
- an argument-exception of the update methods if the type appearing in T_2 is not a supertype of the corresponding type in T_1.

Example 3.7 : Suppose that $Patient$ is a type with 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 8. As $Psychologist$ is not a subtype of $Physician$, the method $doctor_2$ is a return-exception to method $doctor_1$. This exception can cause type errors as shown below. Consider the method $refund(\underline{Hospital}, \underline{Dollar})$ that refunds the expenses of a patient to the hospital he was treated in, and the function $refunding$ that refunds a set of patients using method $refund$.

Figure 8: Doctor and Patient hierarchy

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

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

```
refunding(patients :  PatientSet)
{ foreach 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 $hospital(Psychologist)$. □

Figure 9: Argument-Exception in Component Type Redefinition

$$\begin{tikzcd}
Person & PersonList \\
Student \arrow[u] & StudentList \arrow[u]
\end{tikzcd}$$

$PersonList = list(Person)$
$StudentList = list(Student)$

Example 3.8 : Consider the types in Figure 9. The two representation methods $insert\_element_1(\underline{PersonList}, Person)$ and $insert\_element_2(\underline{StudentList}, Student)$ constitute an argument-exception. □

3.2.3 Structural Consistency and Behavioral Consistency

Figure 10 summarizes the relationships between the two kinds of exceptions to structural consistency and the three kinds of exceptions to behavioral consistency. An arrow from the structural exception x to the behavioral exception y means that x leads to y. For the rest of the report, we only consider exceptions to behavioral consistency, as they also capture exceptions to structural consistency. We define a database schema to be *consistent* iff every method satisfies the behavioral consistency rules.

Figure 10: Subtyping Rules Violations and Exceptions

$$\begin{tikzcd}
\text{Return-Exception} & \text{Argument-Exception} & \text{Explicitly Disallowed Signature} \\
\text{Type Redefinition} \arrow[u] \arrow[ur] & & \text{Data Structure Mismatch} \arrow[u]
\end{tikzcd}$$

3.3 Exceptions to Consistency 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 behavioral consistency is that a correct statement may be unsafe, i.e., yield a type error at run-time. Thus, in presence of exceptions to behavioral consistency, type checking must further partition correct statements into *safe* and *unsafe* statements.

Figure 11: Exceptions to Consistency and Type Errors

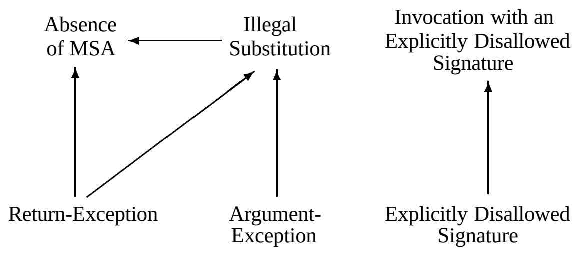

Absence of MSA
Illegal Substitution
Invocation with an Explicitly Disallowed Signature
Return-Exception
Argument-Exception
Explicitly Disallowed Signature

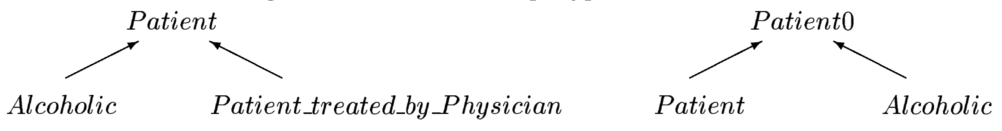
Figure 11 summarizes the relationships between the three different kinds of exceptions to behavioral consistency (bottom of figure) and the three kinds of type errors at run-time (top of figure) : an arrow from x to y means that an exception of kind x may lead to a type error of kind y at run-time.

3.4 Solutions to Avoid Exceptions to Consistency

Object-oriented design offers several solutions to the problems of consistency 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 3.7.

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.

Figure 12: Intermediate Supertype Creation



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

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 12 : (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.

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 some methods. In our previous example, every method that calls $refunding$ 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 $hospital(\underline{Doctor})$, 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 :

```
refunding(patients:PatientSet)
{ foreach 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 $foo(\underline{Patient})$ and $foo(\underline{Alcoholics})$. 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 :  PatientSet)
{ foreach 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.

In conclusion, the painful aspects of these solutions are either the creation of new intermediate types, the addition of new artificial methods, the renunciation of polymorphism by not declaring a type as a subtype of another one, or the intervention of the programmer to test the result of methods that may return NULL values. These modifications are costly in a schema evolution context. Furthermore, they are defined by the user on an ad-hoc basis.

4 The Proposed Solution

Our solution aims at allowing subtyping with exceptions to consistency, while enforcing type safety. In this section, we introduce the *check statements*, that allow to accept unsafe statements due to exceptions while guaranteeing that no type error can occur at run-time. We then show the impact of schema evolution on these check statements. We finally sketch the steps of the type-checking process.

4.1 Check Statements

Check statements embed every statement identified as unsafe at compile-time, as shown in Figure 13. 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.

Figure 13: Check Statements

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

Throughout this report, we consider statements that are either function invocations or variable assignments, as shown in Figure 14. Dynamic type checking involves evaluating their arguments, which may be invocations of functions. Verifying the correctness leads to execute twice these functions, in the condition and unsafe statement parts. In case of functions with side-effects, the second execution is undesirable. To overcome this limitation, sub-expressions of the arguments can be bound to variables local to the CHECK. These variables are untyped (as “void” variables in [KM94] or “dynamics” in [ACPP89]), and can be used both in the unsafe statement part of the CHECK and its exceptional-handling code instead of the original invocations with side-effects, so that these invocations are not evaluated twice[1].

Figure 14: Pseudo-EBNF Grammar of Statements

$$\begin{aligned} statement &::= assignment \mid invocation \\ assignment &::= variable \leftarrow expression \\ invocation &::= function\_name(expression^*) \\ expression &::= variable \mid constant \mid invocation \end{aligned}$$

The condition part is *intensionally* mentionned in the sense that the types for which the exception occurs are not explicitly given. Evaluating the correctness condition involves taking the run-time type of the expressions composing the statement and verifying that the statement is correct with these types, which amounts to query the schema at run-time. Depending on the statement, two expressions of the condition are defined, as shown in Figure 15.

Example 4.1 : In Example 3.1, invocation $build\_abstract(retrieve(myPerson, \text{“}database\ systems\text{”}))$ is unsafe because $myPerson$ may contain a student. Thus, this statement must be surrounded by a CHECK. Let us assume that the generic function $retrieve$ has a side-effect, e.g. it

[1] In the following sections, we assume that only functions without side-effect are used as invocation’s arguments of unsafe statements.

Figure 15: Expression of CHECK Conditions

Unsafe Statement	Condition Part
Invocation $m(e_1, \ldots, e_n)$	m `IS CORRECT ON` $(e_1, \ldots, e_n)$
Assignment $v \leftarrow e$	e `MAY BE ASSIGNED TO` v

increments a counter of users. In order to prevent the increment from happening twice, we shall use a foo variable that stores the result of $retrieve$ in the CHECK condition. Then, variable foo is used in both the unsafe statement and the exception-handling code, as shown in Figure 16. □

Figure 16: CHECK of an Invocation

```
CHECK build_abstract IS CORRECT
         ON foo:=retrieve(myPerson,"database systems")
     build_abstract(foo) ;
ELSE
     introduction(foo) ;
END
```

Some schema evolution operations require to re-evaluate existing programs, which possibly leads to add or delete CHECK statements. Additions are due to newly unsafe statements, and deletions are due to previously unsafe statements became safe. The intensional form spares to reformulate existing CHECK statements retained by the new evaluation.

Example 4.2 : In Example 3.7, 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 as shown in Figure 17. □

Figure 17: Schema Evolution without Generation of a New Check

```
CHECK $hospital$ IS CORRECT ON $(doctor(p))$
     $refund(hospital(doctor(p)), bill(p))$
ELSE
    /* exceptional statement to be provided by the user */
END
```

Example 4.3 : Suppose that a new type of patient, $Tubercular$, is introduced, whose expenses are expressed in Swiss francs (SF). As $bill(p)$ may return Swiss francs, and hospitals may only be refunded Dollars, there exists a signature $(Hospital, SF)$ for which no $refund$ method is applicable. Thus $refund(hospital(doctor(p)), bill(p))$ is unsafe, even when $hospital(doctor(p))$ is safe. As shown in Figure 18, a nested check statement must be generated. □

4.2 Type Checking Process

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 uses 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: indeed, when unsafe subexpressions share some variables or some subexpressions, checks may become redundant. The basic idea to minimize the number of checks is to have the type checker 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 until all the program has been type-checked, so that the user can give, 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.

Figure 18: Schema Evolution with Generation of a New Check

```
CHECK $hospital$ IS CORRECT ON $(doctor(p))$
    CHECK $refund$ IS CORRECT ON $(hospital(doctor(p)), bill(p))$
        $refund(hospital(doctor(p)), bill(p))$
    ELSE
        /* exceptional statement to be provided by the user */
    END
ELSE
    /* exceptional statement to be provided by the user */
END
```

5 Basic Definitions

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

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-targeted systems, the two notions merge, i.e., every target match 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 $equal(Student, Person)$. 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 $equal_2(\underline{Student}, Student)$ 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 "$m_\top$" otherwise. The method $m_\top$ uses a specific "impossible" type, noted $T_\top$, as the type of its arguments and result. $T_\top$ is in strict supertype relation with all other types, i.e., $\forall T,\ T \prec T_\top$. 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.

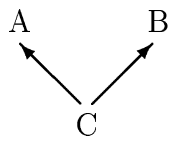
Signature of Expressions. The signature of a tuple of expressions $(e_1 : \tau_1, \ldots, e_n : \tau_n)$ is the tuple $(\tau_1, \ldots,\ \tau_n)$. The signature of a method $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$ is the signature of its formal arguments, i.e., $(T_k^1, \ldots, T_k^n)$. The signature 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 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 $(\tau_1, ..., \tau_n)$. 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 19: Example Schema



$m_1(\underline{A},A)$
$m_2(\underline{B},B)$

$m_2 < m_1$

Example 5.1 Using the type hierarchy in Figure 19, we have :

$$\begin{aligned} &cover(A, \overline{A}) = \{(A, A), (C, A)\} \\ &cover(m_1) = cover(A, A) = \{(A, A), (A, C), (C, A), (C, C)\} \\ &cover(m_2) = cover(B, B) = \{(B, B), (B, C), (C, B), (C, C)\} \quad \square \end{aligned}$$

Well-typed signatures. The well-typed signatures of a generic function m, denoted $well\text{-}typed(m)$, is the union of the covers of all the methods associated with m :

$$well\text{-}typed(m) = \bigcup_{m_i} cover(m_i)$$

Example 5.2 Considering the example of Figure 19, we have :

$$\textit{well-typed}(m) = \{(A,A),(A,C),(C,A),(C,C),(C,B),(B,B),(B,C)\}$$

Intuitively, $\textit{well-typed}(m)$ represents the set of invocation's signatures of m for which there exists a method m_i that is a total match. □

Range of a method. Let m_i be a method for a generic function of arity n, and m a function invocation. The *range* of m_i, noted $range(m_i)$, is the set of signatures for which m_i is the MSA method :

$$\text{range}(m_i) = \{(T_1,\ldots,T_n) \in \textit{well-typed}(m) | \text{ MSA}(m(T_1,\ldots,T_n)) = m_i\}$$

Example 5.3 Considering the example of Figure 19, we have :

- $range(m_1) = \{(A,A),(A,C)\}$
- $range(m_2) = \{(B,B),(B,C),(C,A),(C,C),(C,B)\}$

As the applicability of a method relies on a target match, we take for m_1 (resp. m_2), all signatures (T,T') in $\textit{well-typed}(m)$ such that $T \preceq A$ (resp. $T \preceq B$). Observe that for signatures (C,A), (C,C), and (C,B), m_1 and m_2 are both applicable but since $m_2 < m_1$, these signatures belong to the range of m_2. Finally, note that $(C,A) \in range(m_2)$ but $(C,A) \notin cover(m_2)$. This is a consequence of single-targeting. □

Explicitly Disallowed Signatures of a Method. The set of explicitly disallowed signatures of a method m_i, noted $explicit(m_i)$, is the set of explicitly disallowed signatures of m that belong to the range of m_i.

These signatures are both in the range *and* the cover of m_i, as they correspond to the user's wish to forbid some otherwise type correct invocations. Thus,

$$explicit(m_i) \quad \subseteq \quad cover(m_i).$$

Example 5.4 Let us reconsider the schema introduced in Example 3.4, but for brevity, let the types be P for $Person$, Pr for $Professor$, S for $Student$, TA for $Teaching-Assistant$, R for $Resource$, Sa for $Salary$ and G for $Grant$. Consider the method $allocate_2$. We have :
$range(allocate_2) = \{(Pr,R),(Pr,SA),(Pr,G),(S,R),(S,Sa),(S,G),(T,R),(T,Sa),(T,G)\}$ and $cover(allocate_2) = \{(Pr,Sa),(Pr,G),(T,Sa),(T,G)\}$.
Finally, we can see that $explicit(allocate_2) = \{(T,G)\}$ is included in $cover(allocate_2)$ □

Implicitly Disallowed Signatures of a Method. The set of implicitly disallowed signatures of a method m_i, noted $implicit(m_i)$ is given by :

$$implicit(m_i) = \{(T_1,\ldots,T_n) \in range(m_i) \mid T_{p+1},\ldots,T_n \not\preceq T_i^{p+1},\ldots,T_i^n\}$$

The implicitly disallowed signatures belong to the range of the method but are not covered by it. For invocations with such signatures, the MSA m_i is not a total match. Thus, we also have :

$$\begin{aligned} implicit(m_i) &= range(m_i) - cover(m_i) \\ &= \{(T_1,\ldots,T_n) \in \textit{well-typed}(M) \mid m_i = MSA(m(T_1,\ldots,T_n)) \text{ and } m_i \text{ is not} \\ &\quad \text{a total match for } m(T_1,\ldots,T_n)\} \end{aligned}$$

When all arguments are targetted (i.e., $p = n$), the range of a method m_i is a subset of the signatures covered by m_i. Thus, we have:

Fact 5.1 *If a function m is targetted on all arguments, then $implicit(m_i) = \emptyset$ for all of its methods.*

Example 5.5 Consider again the method $allocate_2$. We have : $implicit(allocate_2) = \{(Pr, R), (T, R)\}$ and we can see that it is equal to $range(allocate_2) - cover(allocate_2)$ $\Box$

As the explicitly disallowed signatures of a method are in its cover, on the contrary to its implicitly disallowed signatures, we have the following fact :

Fact 5.2 $\forall m_i,\ implicit(m_i) \cap explicit(m_i) = \emptyset$

Disallowed Signatures of a Method. The set of *disallowed signatures* of a method m_i, noted $disallowed(m_i)$, is defined as :

$$\forall m_i,\ disallowed(m_i) = explicit(m_i) \cup implicit(m_i)$$

Example 5.6 Applying the above definition to $allocate_2$, we have $disallowed(allocate_2) = \{(T, G), (Pr, R), (T, R)\}$. One can verify in the same way that $disallowed(allocate_1) = \emptyset$ $\Box$

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 3.7, 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 dynamic types 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 $dynamics$.

```
$check(v \leftarrow e)$ /* check for assignments */
input: an assignment $v \leftarrow e$
output: $incorrect$, $safe$ or $unsafe$
**Step 1:**/* Safety w.r.t static types : replace $v$ and $e$ by their static type using $static$ */
    reducedAssignment $\leftarrow$ ( $static(v) \leftarrow static(e)$ ) ;
    result $\leftarrow$ $check_R$( reducedAssignment );
    if result is not safe
        return result ;
**Step 2:**/* Safety w.r.t dynamic types*/
    if $check(e)$ is not safe
        return unsafe ;
    /* Replace the right-hand side by each of its most general dynamic types using $dynamics$ */
    for each $T \in dynamics(e)$ do
        reducedAssignment $\leftarrow$ ( $static(v) \leftarrow T$ ) ;
        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 $m(e_1, \ldots, e_n)$, 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 $m(e_1, \ldots, e_n)$ is respectively incorrect or unsafe. Otherwise the invocation is statically correct 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.

```
$check(m(e_1, \ldots, e_n))$ /* check for invocations */
input: an invocation $m(e_1, \ldots, e_n)$
output: $incorrect$, $safe$ or $unsafe$
**Step 1:**/* Safety with respect to static types : replace arguments by their static type using $static$ */
    reducedInvocation $\leftarrow$ ( $m(static(e_1), \ldots, static(e_n))$) ;
    result $\leftarrow$ $check_R$(reducedInvocation) ;
```

```
  if result is not safe
      return result ;
Step 2:/* Safety with respect to dynamic 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 $s \in signatures(m(e_1, \ldots, e_n))$ do
      reducedInvocation $\leftarrow m(s)$ ;
      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 $m(s) = m(\tau_1, \ldots, \tau_n)$. The type checking of reduced assignments is defined as follows.

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

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

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 :

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

We now give the algorithm to type-check reduced invocations :

```
$check_R(m(s))$
input: a reduced invocation $m(s)$
output: $incorrect$, $safe$ or $unsafe$
  $msa \leftarrow MSA(m(s))$ ;
Step 1 : /* Check the correctness */
  if $msa = m_\top$ or $msa$ is not a total match or $s \in explicit(msa)$
```

```
        return **incorrect** ;
**Step 2 :** /* Check the safety */
    for each $s' \in cover(s)$ do
        $msa' \leftarrow MSA(m(s'))$ ;
        if $msa' = m_\top$ or $msa'$ is not a total match or $s' \in explicit(msa')$
            return **unsafe** ;
    end do ;
    return **safe** ;
    end check
```

6.4 Static and Dynamic Types of an Expression

The static type of an expression can now be defined as shown on Figure 20.

Figure 20: Static Type of Expressions

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

Example 6.1: Consider again the types and methods of Figure 8 in Section 3. Let $refund(\underline{Hospital}, \underline{Dollar})$ be the method used in Example 3.7 to refund 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 21. 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).

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 $m(s)$.

Figure 21: Dynamic Types of an Expression

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 $m(e_1, \ldots, e_n)$	$dynamics(m(e_1, \ldots, e_n)) = max_{\preceq}(\bigcup_{s \in signatures(m(e_1,\ldots,e_n))} dynamics(m(s)))$

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 $hospital(doctor(p))$.

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

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 Optimizing the Type Checking of Reduced Invocations

In this section, we propose an optimization of the type-checking of reduced invocations. The algorithm for $check_R$ presented in section 6.3 is expensive because it requires to compute the MSA for every signature in the cover of the reduced invocation. Optimizing $check_R$ is particularly important as it is called several times by $check$ to type check a general invocation. The idea of the optimization is the following. Given a reduced invocation, if no signature in its cover, i.e. the run-time signatures, is a disallowed signature of some method, then the invocation is safe. To evaluate this condition, one computes the set of disallowed signatures of the methods that are more specific than the MSA of the reduced invocation. This set is called the *potential disallowed signatures* of the MSA. If a method has no potential disallowed signatures, then all invocations for which it is the MSA are safe. Such a property of a method is called *static safety* and constitutes a cheap sufficient condition for the safety of a reduced invocation.

In the following, we first give the optimized algorithm, and then present the two safety conditions that it uses.

7.1 Optimized Algorithm for the Static Type Checking of Reduced Invocations

The first step of the optimized algorithm checks the correctness of a reduced invocation, following the same criteria as in 6.3 on the MSA of the invocation. Steps 2 and 3 check the safety. Step 2 checks whether the MSA of the invocation is static safe. If it is not, step 3 verifies that no run-time signature, i.e. no signature covered by the invocation, is a potential disallowed signature of the MSA.

```
$check_R(m(s))$
input: a reduced invocation $m(s)$
output: $incorrect$, $safe$ or $unsafe$
    $msa \leftarrow MSA(m(s))$ ;
**Step 1 :**
    if $msa = m_\top$ or $msa$ is not a total match or $s \in explicit(msa)$
        return **incorrect** ;
**Step 2 :**
    if $msa$ is static safe
        return **safe** ;
**Step 3 :**
    if no signature in $cover(m(s))$ is a potential disallowed signature of $msa$
        return **safe** ;
    return **unsafe** ;
    end check
```

7.2 Safety and Potential Disallowed Signatures

The third step of the algorithm relies on Proposition 1 below, that gives a necessary and sufficient condition for the safety of a correct reduced invocation.

Potential Disallowed Signatures of a Method. The set of potential disallowed signatures of a method m_i, noted $potential\_disallowed(m_i)$, is defined as :

$$potential\_disallowed(m_i) = \bigcup_{m_j \leq_s m_i s \in cover(m_i)} disallowed(m_j)$$

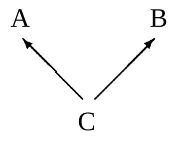
Proposition 1 *A correct reduced invocation $m(s)$ is safe iff*

$$potential\_disallowed(MSA(m(s))) \cap cover(s) = \emptyset$$

Proof : see Appendix A. □

Example 7.1 Consider the hierarchy and methods of Figure 22. The potential disallowed signature of m_1 is (C, A), and there is no potential disallowed signature for m_2. Invocation $m(A, A)$ is declared unsafe, because the signature (C, A) is covered by it. □

Figure 22: Safety Conditions



$m_1(\underline{A},\underline{A})$
$m_2(\underline{B},\underline{B})$

$m_2 < m_1$

$explicit(m_1)$=(C, A)

7.3 Static Safety of a Method

The second step of the algorithm relies on Proposition 2 below, that gives a sufficient condition for the safety of a correct reduced invocation. It uses the notion of *static safety* of a method. This property is invocation-independent and may be computed once for each method, at compile-time.

Static safety of methods. A method m_i is *static safe* iff

$$potential\_disallowed(m_i) \cap cover(m_i) = \emptyset$$

As $\forall s,\ cover(s) \subseteq cover(MSA(m(s)))$, we have :

Proposition 2 *$MSA(m(s))$ is static safe $\Rightarrow$ $m(s)$ is safe.*

Proof : As $cover(s) \subset cover(MSA(m(s)))$, we have $potential\_disallowed(MSA(m(s))) \cap cover(MSA(m(s))) = \emptyset \Rightarrow potential\_disallowed(MSA(m(s))) \cap cover(s) = \emptyset$, which implies the safety from Proposition 1. □

Example 7.2 In Figure 22, the method m_2 has no potential disallowed signature, thus it is static safe. The invocation $m(A, C)$ is safe even though its MSA, m_1, is not static safe, and $m(B, C)$ is safe. □

8 Safety and Consistency

In this section, we establish the relation between safety and consistency, introducing the notion of *trespassing* method. A method m_i trespasses on method m_j, if m_i may be selected at run-time for invocations whose MSA at compile-time is m_j. We show that argument-exceptions and return-exceptions cause safety problems only if they are coupled with trespassing. If m_i is an argument- or return-exception to m_j, but may not be selected for invocations whose MSA at compile-time is m_j, then no run-time type error may occur.

Example 8.1 Consider the schema of Figure 23 with method $foo_1(E) \rightarrow E$. The MSA of invocation $m(a, d)$ is m_1. As method m_2 is both a return-exception and an argument-exception to m_1, a run-time type error could occur if m_2 was selected at run-time. For example, $m(a, d)$ could return a D and there would not be any foo method applicable to the invocation $foo(m(a, d))$. However, all invocations for which both m_1 and m_2 are applicable, namely $m(C, D)$ and $m(C, E)$, have m_3 as their MSA. And m_3 is consistent w.r.t. m_1 and m_2. Thus, the inconsistency of m_2 w.r.t. m_1 cannot lead to any safety problem, because m_2 never trespasses on m_1. □

Figure 23: Consistent Schema

$$\begin{tikzcd}
A & & B & D \\
& C \arrow[ul] \arrow[ur] & & E \arrow[u]
\end{tikzcd}$$

$m_1(\underline{A},D) \rightarrow E$	$m_2 < m_1$
$m_2(\underline{B},E) \rightarrow D$	$m_3 < m_2$
$m_3(\underline{C},D) \rightarrow E$	$m_3 < m_1$

We first give the definition of a trespassing method, then a proposition that states the relationship between trespassing, consistency and safety.

Trespassing. A method m_i *trespasses* on method m_j iff:

$$range(m_i) \cap cover(m_j) \neq \emptyset$$

Example 8.2 In the example of Figure 24, (C, A) and (C, C) are both in $range(m_2)$ and in $cover(m_1)$. Thus, m_2 trespasses on method m_1. □

Figure 24: Trespassing

$$\begin{tikzcd}
\mathrm{A} & & \mathrm{B} \\
& \mathrm{C} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

$m_1(\underline{A},A)$
$m_2(\underline{B},B)$

$m_2 < m_1$

Proposition 3 *If m_i is a return- and/or argument-exception to m_j, then a run-time type error may occur only if m_i trespasses on m_j.*

Proof : see Appendix B. □

Example 8.3 Going back to Figure 23, we see that there is no method m_i that is both an argument-exception to some method m_j and trespasses on it : m_2 is an argument-exception to m_1, but does not trespass on it, while m_3 trespasses on both m_1 and m_2, but is not an argument-exception to them. Thus, $implicit(m_i) = \emptyset$ for all m_i. Note that removing m_3 makes m_2 trespassing on m_1, so that $implicit(m_2) = \{(C, D)\}$.

Moreover, there is no method m_i that is both a return-exception to some method m_j and trespasses on it : m_2 is a return-exception to m_1, but does not trespass on it, while m_3 trespasses on both m_1 and m_2, but is not a return-exception to them. Thus, the type returned at run-time by any well-typed invocation $m(s)$ is guaranteed to be a subtype of the static type of $m(s)$. □

It must be noted that although the covariance and contravariance rules are too pessimistic, they are adopted in most systems because they are simpler to check and offer a better support for schema evolution. Indeed, adding or removing a method that abides to the covariance and contravariance rules with respect to all other methods, has no consequences on safety. As we showed in the above examples, this is not the case when trespassing is taken into account.

9 Final Steps

The last two steps of the type checking process are the generation of check statements and the type checking of the exception-handling code provided by the user. These two issues are out of the scope of this report. In this section, we just give an idea of the problems and sketch the solution.

9.1 Generation of Check Statements

As invocations may appear as arguments of other invocations, a single statement may contain several unsafe subexpressions. This naturally leads to nest the check statements.

Example 9.1: We reuse the methods and types of Figure 8 and assume there exists the following methods : $bill(Patient) \rightarrow Dollar$, to get the expenses of a patient and $refund(\ Hospital, Dollar)$ to refund hospitals the expenses of their patients. Moreover, we assume that alcoholics are not billed for their treatment, i.e., $Alcoholic$ is an explicitly disallowed signature of $bill(Patient)$. Consider statement $refund(hospital(doctor(p)), bill(p))$. A naive approach to check generation examines each subexpressions of a statement in a left to right, depth first order and generates a check statement whenever an unsafe subexpression is encountered. Thus, after type checking p,$doctor(p)$, $hospital(doctor(p))$, p, $bill(p)$, the following nested check statements have been generated :

```
CHECK $hospital$ IS CORRECT ON $(doctor(p))$
  CHECK $bill$ IS CORRECT ON $p$
  ELSE
  END
ELSE
END
```

However, the check on $bill(p)$ is redundant. Indeed, if $hospital(doctor(p))$ is correct, then $doctor(p)$ is a $Physician$ and p is not an $Alcoholic$. Thus, $bill(p)$ is safe. □

To correctly generate check statements, the idea is to provide the type checker with the ability to infer the run-time types of sub-expressions like $doctor(p)$ and p, based on the previously generated checks. The inferred types are then used to bind the remaining subexpressions, using what can be called *type closures*. These bindings are then used by the type checker.

9.2 Type Checking the Exception-Handling Code

Type checking the exception-handling code provided by the user differs from the type checking we defined in the two previous sections. To give an idea of the problem, consider the following example :

Example 9.2: Going back to the doctor and patient hierarchy of Figure 8, assume there exists a method $practice(Psychologist) \rightarrow Office$ to access the practice of psychologists and two methods $address(Hospital) \rightarrow Address$ and $address(Office) \rightarrow Address$ to get the addresses. Consider the exception-handling code of the following check statement :

```
$addr\ :\ Address$;
CHECK $hospital$ IS CORRECT ON $(doctor(p))$
  $addr \leftarrow address(hospital(doctor(p)))$;
ELSE
  $addr \leftarrow address(practice(doctor(p)))$;
END
```

In the ELSE part, one can infer that $doctor(p)$ is of type $Psychologist$. This allows to write a modified version of the original statement using $practice$ instead of $hospital$. However, note that the ELSE statement is not correct according to standard static type-checking, as $static(doctor(p)) = Physician$ and $practice$ is not applicable to $Physician$. □

As for the generation of check statements, the solution is to provide the type checker with the ability to infer the run-time types of subexpressions like $doctor(p)$ based on the previously generated checks.

10 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, DS92] 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).

Bounded type quantification, first introduced in [CW85], appears in several proposals [CM92, CCHO89, KM94] to extend the flexibility of statically typed object languages. As explained in [KM94], it enables "polymorphic operations [...] to deal with objects of different types that do not necessarily lie on the same branch of the super/subtype relationship". [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. Finally, F-bounded quantification [CCHO89] allows to support recursively defined types, like $Person$ and $Student$ in Figure 1.

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-targeted methods and recursion. Covariant updates are shown to maintain consistency.

[MMMP90] recognizes the conflict that arises from the use of the type system both "as a means for representing concepts in the application domain and for detecting [...] type errors". They show that the subtyping of "virtual classes" (i.e., classes with a type parameter) introduces type holes, similar to component type redefinition. They conclude that a combination of compile-time and run-time type checks, as implemented in Beta, gives a good balance of flexibility and type safety. All operations on virtual classes involve run-time type checking. Furthermore, an error occurs if a statement in a Beta program is not correct at run-time.

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-targeted 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.

11 Conclusion

In this report, 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 covariance, 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 modifications 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-targeted functions.

All the steps of the proposed type checking process have now been specified (see [Ami94]). 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 addresses 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 report. We also would like to thank the three referees for their extremely careful reviews and helpful comments.

12 Appendices

A Proof of Proposition 1

We first introduce he following lemma :

Lemma 12.1 $\forall s, s',\ s' \in cover(s) \Rightarrow MSA(m(s')) \leq_{s'} MSA(m(s))$

Proof : As $s' \preceq s$, $MSA(m(s))$ is applicable to s'. As $MSA(m(s'))$ is the most specific method applicable to s' w.r.t. the ordering of s', $MSA(m(s'))$ is more specific or equal to $MSA(m(s))$. $\square$

We prove that a necessary and sufficient condition of safety on $m(s)$ is that $potential\_disallowed(MSA(m(s))) \cap cover(s) = \emptyset$. This amounts to the following equivalence :

$$m(s) \text{ is correct and } \Big(\bigcup_{m_i \leq_{s''} MSA(m(s)), s'' \in cover(MSA(m(s))} disallowed(m_i)\Big) \cap cover(s) = \emptyset \tag{1}$$

equivalent to

$$\begin{aligned} &\forall s' \in cover(s),\ MSA(m(s')) \neq m_\top \text{ and} \\ &\quad MSA(m(s')) \text{ is a total match for } m(s') \text{ and } s' \notin explicit(MSA(m(s'))) \end{aligned} \tag{2}$$

Using the definition of the correctness of a reduced invocation, (1) can be written :

$$\begin{aligned} &MSA(m(s)) \neq m_\top \text{ and } MSA(m(s)) \text{ is a total match for } m(s) \text{ and} \\ &\quad \forall s' \in cover(s),\ \forall s'' \in cover(MSA(m(s))),\ \forall m_i \leq_{s''} MSA(m(s)), s' \notin disallowed(m_i) \end{aligned} \tag{3}$$

To rewrite (2) we use the following equivalence, that comes from the definition of $implicit(m_i)$:

$$MSA(m(s')) \text{ is a total match for } s' \Leftrightarrow s' \notin implicit(MSA(m(s')))$$

Thus (2) can be written :

$$\forall s' \in cover(s),\ MSA(m(s')) \neq m_\top \text{ and } s' \notin disallowed(MSA(m(s'))) \tag{4}$$

Let us prove now that (3) $\Rightarrow$ (4). We assume that (3) is true for m and s. $MSA(m(s)) \neq m_\top$ implies $\forall s' \in cover(s),\ MSA(m(s')) \neq m_\top$. Thus the first conjunct of (4) is established.

We also prove that $\forall s' \in cover(s),\ s' \notin disallowed(MSA(m(s')))$, applying the second conjunct of (3). From lemma 12.1, we have $MSA(m(s')) \leq_{s'} MSA(m(s))$, and $s' \in cover(s)$ implies $s' \in cover(MSA(m(s)))$. Thus $s' \notin disallowed(MSA(m(s')))$. This concludes the first part of our proof.

Let us prove now that (4) $\Rightarrow$ (3). We assume that (4) is true for m and s. As $s \in cover(s)$, we have that $MSA(m(s)) \neq m_\top$. As $s \notin implicit(MSA(m(s)))$, $MSA(m(s))$ is a total match for $m(s)$.

Now let $s' \in cover(s)$, $s'' \in cover(MSA(m(s)))$, and $m_i \leq_{s''} MSA(m(s))$. If $m_i = MSA(m(s'))$, from (4) we have $s' \notin disallowed(m_i)$. If $m_i \neq MSA(m(s'))$, from the definition of $implicit(m_i)$ and $explicit(m_i)$, $disallowed(m_i) \in range(m_i)$, thus $s' \notin disallowed(m_i)$. This concludes our proof of proposition 1.

B Proof of Proposition 3

We prove that a run-time type error due to a method m_i being a return- or an argument-exception to a method m_j, may occur only if m_i trespasses on m_j. We first consider the case of return-exceptions, then of argument-exceptions.

A run-time type error may occur due to a method m_i being a return-exception to a method m_j, iff for some static signature $s \in \mathit{well\text{-}typed}(m)$, $m_i \in RTC(m(s))$ and $m_j = MSA(m(s))$. We have to prove that in this case, m_i trespasses on m_j.

As $m_i \in RTC(m(s))$, there exists $s' \in \mathit{cover}(s)$ such that $m_i = MSA(m(s'))$. As $s \in \mathit{cover}(m_j)$, we have $s' \in \mathit{cover}(m_j)$, and $s' \in \mathit{range}(m_i)$, thus $\mathit{range}(m_i) \cap \mathit{cover}(m_j) \neq \emptyset$. This concludes the first part of our proof.

Let us now consider the case of argument-exceptions. The general static safety condition is :

$$\mathit{potential\_exceptions}(m_i)) \cap \mathit{cover}(m_i) = \emptyset$$

Using the decomposition of exceptions into implicit and explicit exceptions, we can rewrite the condition as :

$$((\bigcup_{m_j \leq m_i} \mathit{implicit}(m_j)) \cup (\bigcup_{m_j \leq m_i} \mathit{explicit}(m_j))) \cap \mathit{cover}(m_i) = \emptyset \Leftrightarrow$$

$$(\bigcup_{m_j \leq m_i} \mathit{implicit}(m_j)) \cap \mathit{cover}(m_i) = \emptyset, \text{ and } (\bigcup_{m_j \leq m_i} \mathit{explicit}(m_j)) \cap \mathit{cover}(m_i) = \emptyset$$

Static safety w.r.t. implicit exception corresponds to the first part of the conjunction. As the implicit exceptions of methods are due to argument exceptions, let us see what conditions must hold on m_i and the $m_j < m_i$.

As $\mathit{implicit}(m_i) \cap \mathit{cover}(m_i) = (\mathit{range}(m_i) - \mathit{cover}(m_i)) \cap \mathit{cover}(m_i) = \emptyset$, the first conjunctive term can be written as:

$$\bigcup_{m_j < m_i} \mathit{implicit}(m_j) \cap \mathit{cover}(m_i) = \emptyset \tag{5}$$

Let us prove that if (5) is false, then :

$\exists m_j < m_i$ such that m_j is an argument-exception to m_i and trespasses on m_i

For this, we show that if either (A) m_j is not an argument-exception to m_i, or (B) m_j does not trespass on m_i, then (C) $\mathit{implicit}(m_j) \cap \mathit{cover}(m_i) = \emptyset$.

Let us first prove (A) $\Rightarrow$ (C). (A) means that :

$$T_j^{p+1}, \ldots, T_j^n \succeq T_i^{p+1}, \ldots, T_i^n \tag{6}$$

For all $T^1, \ldots, T^n$ in $\mathit{implicit}(m_j)$, we have :

$$T^{p+1}, \ldots, T^n \not\preceq T_j^{p+1}, \ldots, T_j^n \tag{7}$$

(6) and (7) imply :

$$T^{p+1}, \ldots, T^n \not\preceq T_i^{p+1}, \ldots, T_i^n$$

And thus :

$$m_j \text{ not argument-exception to } m_i \Rightarrow \mathit{implicit}(m_j) \cap \mathit{cover}(m_i) = \emptyset$$

Let us now prove (B) $\Rightarrow$ (C). We have that $\mathit{implicit}(m_j) = \mathit{range}(m_j) - \mathit{cover}(m_j)$ and (B) means that $\mathit{range}(m_j) \cap \mathit{cover}(m_i) = \emptyset$. Thus, $\mathit{implicit}(m_j) \cap \mathit{cover}(m_i) = \emptyset$. This concludes the proof.

References

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. POPL*, Austin,TX, January 1989.

[ADL91] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static type checking of multi-methods. In *Proceedings of the sixth International Conference on Object-Oriented Programming : Systems, Languages and Applications*, Phoenix, AZ, October 1991.

[AKW90] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas. In *Proceedings of the fourth ACM Symposium on the Principles of Database Systems*, Nashville, MA, April 1990.

[Ami94] Eric Amiel. *Schémas orientés-objet : Exceptions à la cohérence Comportementale et Envoi de Multi-Méthodes*. PhD thesis, Université Paris VI, 1994.

[App94] Apple Computer. *Dylan Interim Reference Manual*, June 1994. Available by ftp from ftp.cambridge.apple.com in /pub/dylan/dylan-manual.

[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*, 23(9), 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 *Proceedings of the first International Conference on Object-Oriented Programming : Systems, Languages and Applications*, Portland, OR, September 1986.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management Of Data*, San Francisco, CA, June 1987.

[Bor88] A. Borgida. Modeling class hierarchies with contradictions. In *Proceedings of the ACM SIGMOD International Conference on Management Of Data*, Chicago, IL, June 1988.

[Bor89] A. Borgida. Type systems for querying class hierarchies with non-strict inheritance. In *Proceedings of the eight ACM Symposium on Principles of Database Systems*, Philadelphia, PA, March 1989.

[Bru93] K. B. Bruce. Safe type-checking in a statically-typed object-oriented programming language. In *Proc. POPL*, January 1993.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Proceedings of the Symposium on Semantics of Data Types*, 1984. In LNCS 173.

[Cat94] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kauffman, 1994.

[CCHO89] P. Canning, W. Cook, W. Hill, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. Intl. Conf. on Functional Programming and Computer Architecture*, London, United Kingdom, September 1989.

[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 *Proceedings of the eleventh International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara,CA, August 1993.

[Cha92] C. Chambers. Object-oriented multi-methods in Cecil. In *Proceedings of the sixth European Conference on Object-Oriented Programming*, Utrecht, Netherlands, June 1992.

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

[CMM91] R.C.H. Connor, D. McNally, and R. Morrison. Subtyping and assignment in database programming languages. In *Proceedings of the second International workshop on Database Programming Languages*, Gleneden beach, OR, August 1991.

[Coo89] W. Cook. A proposal to make Eiffel type-safe. In *proceedings of the third European Conference on Object-Oriented Programming*, Nottingham, Great-Britain, July 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 INRIA, editor, *Proceedings of the sixth Journées Bases de Données Avancées*, Montpellier, France, September 1990.

[DCL$^+$93] L. G. DeMichiel, D. D. Chamberlin, B. G. Lindsay, R. Agrawal, and M. Arya. Polyglot: Extensions to relational databases for sharable types and functions in a multi-language environment. In *Proc. Intl. Conf. on Data Engineering*, Vienna, Austria, April 1993.

[DS92] Scott Danforth and Eric Simon. *The Next Generation of Information Systems - from Data to Knowledge*, chapter A Data and Operation Model for Advanced Database Systems. Springer Verlag, 1992.

[FM94] A. Formica and M. Missikoff. Correctness of isa hierarchies in object-oriented database schemas. In *Proc. EDBT*, Cambridge, March 1994.

[KM94] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Prentice-Hall, 1994.

[Laj93] J. Lajoie. The new language extensions. *C++ Report*, 5(6):47–52, 1993.

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

[Mel94] Jim Melton, editor. *(ISO Working Draft) SQL Persistent Stored Modules (SQL/PSM)*. ANSI X3H2-94-331, August 1994.

[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 *Proceedings of the fifth European Conference on Object-Oriented Programming*, Geneve, Switzerland, July 1991.

[MMMP90] O. L. Madsen, B. Magnusson, and B. Moller-Pedersen. Strong typing of object-oriented languages revisited. In *Proc. ECOOP-OOPSLA*, 1990.

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

[Wal91] E. Waller. Schema updates and consistency. In *Proceedings of the second International Conference on Deductive and Object-Oriented Databases*, Munich, Germany, December 1991. In LNCS 566.

[ZM89] S. Zdonik and D. Maier. *Fundamentals of object-oriented databases*, chapter Fundamentals of object-oriented databases, pages 1–32. Morgan-Kaufman, 1989.

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399