Type Safety for Versioned Object-Oriented Programs

Xue Li
School of Computing Science
Queensland University of Technology
GPO Box 2434 Brisbane, Australia
xueli@fit.qut.edu.au

February 14, 1996

Abstract

An object-oriented program is considered as a set of versioned methods. The type safety for versioned object-oriented programs is a problem to prevent run-time type errors. In this paper, polymorphism as a key feature in object-oriented programming will be exploited for type safety of versioned methods. We propose a two-level binding process for the execution of versioned methods. At the first level, the process focuses on the resolution of classical method overriding, overloading, and migration under the assumption of the type conformance between formal and actual parameters. At the second level, the version tree of a method is searched for a type-safe version in the sense of type consistency (sub-typing) between different versions of a method in order to achieve type safety for the execution of versioned methods.

1 Introduction

The type safety is a problem about how to avoid run-time type errors after updates of programs. The updates may involve the creation of versions of programs. This paper address the problems of type safety for the versioned programs.

In object-oriented programming, a program consists of methods (procedures). A method as a property of a class can be re-defined along the class inheritance hierarchy (called overriding) or have multiple implementations within the same lexical scope (called overloading). Polymorphism provides the ability to differentiate object instances that can respond to the same method (called binding, or late binding because this is performed at run time) [14]. However in the context of versioning, where a method may have different versions, polymorphism has to be extended to have its new meaning to handle the method versioning.

The complexity in dealing with the method evolution is that when a method is versioned, a two-level control may need to be applied. At the first level, the binding of a polymorphic method is determined by the mechanism that follows the principles of late binding. At the second level, the binding of a versioned method is subject to an access mechanism to determine which version of the method should be bound.

This problem indicates a distinction should be made between the semantics of binding polymorphic methods and binding versioned methods. The former is about the decision to be made in terms of the types of the binding objects. The latter is about the selection to be performed in terms of the versions that are consistent with the binding objects.
To our knowledge, the two issues of type safety and method versioning have always been addressed separately in literature[1, 12, 14]. A clear understanding on polymorphic methods can be found in [1, 3, 4]. The research provided by Osborn in [13] has addressed the problem of polymorphism in the presence of type updates. Polymorphism has been considered as a mechanism to enforce query equivalence amongst the versions of a class. In [11], polymorphism is embodied in a propagation pattern approach, in which a refinement process is introduced to specify a method from its abstract form represented in a propagation pattern into its language-specific form. The benefit of this approach is for the method reuse and the incremental design. Whenever the structure of a schema is changed, the pre-designed methods will be still able to operate because the propagation pattern in its abstraction form can overlook the structural (incremental) differences of the schema. However, as for the other approaches in versioning (e.g., [8, 12]), the key aspects of polymorphism, such as the binding of versioned methods, have not been investigated. In this context, for example, a polymorphic method may be defined for drawing a colored box or a circle on the screen depending on what method-calling is sent to it. The definition of method can be overloaded by drawing pictures either in x-y coordinates or in polar coordinates. And more, if the method has two versions involving the drawing of 2-Dimension or 3-Dimension pictures, the binding of type-safe versions will be a complex task involving the binding of the correct method (a box or a circle, in x-y, or in polar system) and the right version (in 2-Dimension or in 3-Dimension). So in this case, even for one method-calling, there are $2 \times 2 \times 2 = 8$ possible candidate methods to be bound with. the binding process has to consider not only the situations of method overriding and overloading but also the navigation of versions. Strategies for this kind of binding problem have not been seen from the literature. It would be interesting to see a solution that can be incorporated into a transparent versioning mechanism for binding versioned methods.

2 Method Versioning

A version records information about modification of either an original method or an existing version of a method. Whenever a method is modified, the operation and effects of the operation are to be recorded. To this end, a version tree is resulted for each of the methods.

When a method is invoked, the late binding is a process that decides which of the overloaded, overridden, or migrated methods is to be bound at run time. However this is not sufficient since the binding of the method to a specific version is required. This type of binding is called version binding.

Before we give the criteria and details about the version binding, we clarify some semantic differences among a few concepts.

Migration vs versioning. Semantically, method migration[10] is different from the method versioning. The former is a replacement of a method by another specialized or generalized method while the latter is the change on a method and thus making another state (version) of the method. The method migration happens where a method is bound. It reflects the flexibility of the method-calling (message-sending). While method versioning is a way of varying a method to cope for different applications or designs.

Migration vs overriding. The method migration is distinct from the method overriding. The difference is that the former is a way of method reuse while the latter is a way of method redefinition. The method reuse is based on the method interfaces which satisfy
either covariance or contravariance rules[7]. The method redefinition is based on the
inheritance hierarchy. Note that neither method generalization nor method specialization
is a case of the method overriding because they are defined in different context. The
method specialization/generalization (method migration) is considered in terms of the
interfaces of the methods while method overriding is considered in terms of the class
inheritance. In other words, a method defined in a sub-class can override a method in the
super-class only if they have the same name. There is no sub-typing obligation regarding
the types of overridden method interfaces. The detailed discussions of the differences
between the class inheritance and the method interface inheritance can be found in [5, 6].

The following subsections cover the discussions on the criteria and the scenario of the
version binding.

2.1 Method versions

When a method is updated a method version can be created[9]. The following definitions are
used to define method versions and the structures thereof. Three concepts, namely method
version, method version tree, and method version poset will be introduced.

**DEFINITION 1** (Method version)
Given a method m, the version of m is a triplet \((n_m, o_m, i_m)\), where \(n_m\) is the version identifi-
cation number, \(o_m\) is the method-changing operation, applied on method m or a version of m,
and \(i_m\) represents a method interface.

All versions of a method form a tree structure, namely, a version tree that is used to cluster
the allowable modifications of a method.

**DEFINITION 2** (Method version tree)
A version tree of a method m, denoted as \(T_m\), is a cluster of all m's versions. \(T_m\) is defined as
\(<r, R>\), where

- \(r = (m, m_{inv})\) is the root: given \(m_{inv}\) is the invariant set which records the signature of
  method m, and
- \(R = (V_m, \leq_{mv})\): given \(V_m\) is a set of versions and \(\leq_{mv}\) is the partial order relationship
  between method versions. This relationship is defined as follows: if \(m_1 \leq_{mv} m_2\) for \(m_1\),
  then \(m_2 \in V_m\) is defined such that \(m_2\) is changed from \(m_1\).

Every method may have a partially ordered set (poset) of versions. An element of a poset
holds the definition of a version.

**DEFINITION 3** (Method version poset)
Let \(T_m\) be a version tree \(T_m = <r, R>\) of a method m. The poset of \(T_m\) is defined as \(\text{Poset}_T_m = \{(m) \cup V_m, \varphi\}\) where \(\varphi\) is a pre-order traversal over \(T_m\).

The significance of defining the version poset is to provide a way of linear searching on the
method versions.
2.2 Version binding

The version binding is a process to determine which version of a method is the method to be executed. The criteria of the decision are established according to the type conformity of the actual and formal parameters of the method versions. Before a cluster of versions is searched for the binding purpose, the method polymorphism needs to be considered to determine whether a method is overloading, overriding, or migrating. So, the process of version binding is carried out in two dimensions: Firstly, it is a process to resolve the method overloading and overriding plus method migration situations. Secondly, for a specific method meant to be bound, a right version of the method is determined (or for a given version the type safety is to be decided). In order to determine such a version of a method to be bound with respect to type-safety (or for a given version the type safety is to be decided), we classify the version binding into two categories of relationships that exist between signatures of versions, namely, covariance version sub-typing and contravariance version sub-typing. We will demonstrate various situations that correspond to the application of covariance and contravariance rules to the versions of a method.

2.2.1 Covariance version binding

Covariance version binding is the version binding considered in terms of covariance sub-type rule[2, 5]. The covariance rule says that a method is redefined by specializing the both input and output types.

Let us consider a given method \( m_1 \) and two versions \( m_1^1 \) and \( m_1^2 \) with the signatures \( S_1 \rightarrow t_1, \ S_1^1 \rightarrow t_1^1 \), and \( S_1^2 \rightarrow t_1^2 \) respectively, where \( S \) denotes the input type and \( t \) denotes the output type. A general situation to be considered is that \( m_1 \) satisfies the covariance sub-type rule with one of its versions or a \( m_1 \)’s version satisfies the covariance sub-type rule with \( m_1 \). So, we assume that \( m_1 \preceq_{cv} m_1^1 \) and \( m_1^2 \preceq_{cv} m_1 \), where \( \preceq_{cv} \) denotes the covariance sub-typing. The type-safety of the binding of \( m_1 \) is considered in two cases:

- \( m_1^1 \) can be specialized to \( m_1 \). Therefore \( m_1^1 \) may be bound whenever \( m_1 \) is requested.
- \( m_1 \) can be specialized to \( m_1^2 \). Therefore \( m_1^2 \) may be bound whenever \( m_1 \) is requested.

In the case that a version can be specialized to \( m_1 \) or \( m_1 \) can be specialized to a version, the selection of a version to be bound is arbitrary in the sense of type-safety.

For the discussion, we introduce a format of binding expression: \( m_1 \circ m_2@m_3 \), where \( m_1 \), \( m_2 \), and \( m_3 \) denote the method \( m \) in the binding process. From left to right the semantics of the expression is that the first position indicates a call to a method with actual parameters, the second position indicates the late binding of a method with formal parameters (“@” denotes for late binding), and the third position indicates the one that either \( m_1 \) or \( m_2 \) is redefined from (“@” denotes for method redefinition)[15].

The following table summarizes the type-safety problems in various situations.

Table 1 shows all possible situations during the covariance version binding. The interpretation of these situations is as follows. Example 1 is given to illustrate these situations.

1. \( m_1 \circ m_1@m_1 \): This is the situation that no version is involved. The method-calling to \( m_1 \) is received by \( m_1 \) and furthermore \( m_1 \) is the one actually bound.
<table>
<thead>
<tr>
<th>Case</th>
<th>Version binding</th>
<th>Type safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>( m_1 \circ m_1 @ m_1 )</td>
<td>safe</td>
</tr>
<tr>
<td>(2)</td>
<td>( m_1 \circ m_1 @ m_1 )</td>
<td>safe</td>
</tr>
<tr>
<td>(3)</td>
<td>( m_1 \circ m_1 @ m_1 )</td>
<td>safe</td>
</tr>
<tr>
<td>(4)</td>
<td>( m_1 \circ m_2 @ m_1 )</td>
<td>run-time type error</td>
</tr>
<tr>
<td>(5)</td>
<td>( m_1 \circ m_2 @ m_1 )</td>
<td>compile-time error</td>
</tr>
</tbody>
</table>

Table 1: Covariance version binding

(2) \( m_1 \circ m_1 @ m_1 \): The method-calling to \( m_1 \) is received by \( m_1 \) and \( m_1 \) is the one actually bound. This indicates a method migration that \( m_1 \) is specialized to \( m_1 \). This is a general case of type-safe method versioning where a transparent versioning mechanism is used. This situation also reflects a perfect method reuse that \( m_1 \) is replaced by \( m_1 \).

(3) \( m_1 \circ m_1 @ m_1 \): The method-calling to \( m_1 \) is received by \( m_1 \) and \( m_1 \) is bound where a version of \( m_1 \) namely, \( m_1 \) is expected. This indicates a method migration that \( m_1 \) is specialized to \( m_1 \) however \( m_1 \) is meant to be bound. This situation happens when the method evolution causes that and \( m_1 \) are not able to be bound to the method-calling sender (i.e., the one with the formal parameters) except for \( m_1 \) (e.g., \( m_1 \) is deleted).

(4) \( m_1 \circ m_1 @ m_1 \): The method-calling to \( m_1 \) is received by \( m_1 \) and \( m_1 \) is bound. Comparing to situation 2, this is a general case of type-unsafe method versioning.

(5) \( m_1 \circ m_1 @ m_1 \): The method-calling to \( m_1 \) is received by \( m_1 \) and \( m_1 \) is bound. This means that \( m_1 \) is replaced by \( m_1 \). A compile-time type error would be issued for this case.

In fact, the above discussions treat \( m_1 \) and \( m_1 \) as generic representations for any versions of \( m_1 \). The situations 4, 5 are classified as type-unsafe for the reason that type conformity between interfaces of versions does not satisfy covariance sub-type rule. This actually concludes that a method cannot be specialized if it is a method-calling sender. An intuitive explanation is that a general method is unable to provide information for a specific one. An illustration can be seen in example 1. Above discussions also imply a search for a certain version from the method-version tree in terms of the interfaces of versions that satisfy the covariance sub-type rule. The criteria will be provided at the end of this section.

Table 1 above does not cover other cases that correspond to the permutations of the expressions: \( m_1 \circ m_1 @ m_1 \), \( m_1 \circ m_1 @ m_1 \), \( m_1 \circ m_2 @ m_1 \), and \( m_1 \circ m_1 @ m_1 \) because they are not possible.

**EXAMPLE 1** Let us consider a method \( m_1 \) and its versions \( m_1 \) and \( m_1 \) with signatures of PhDStudent \( \rightarrow \) Lecturer, Student \( \rightarrow \) Staff, and PartTimePhDStudent \( \rightarrow \) PartTimeLecturer respectively. We assume that \( m_1 \leq_{co} m_1 \) and \( m_1 \leq_{co} m_1 \) for the facts of interfaces of PhDStudent \( \leq \) Student, Lecturer \( \leq \) Staff, and PartTimePhDStudent \( \leq \) PhDStudent, and PartTimeLecturer \( \leq \) Lecturer (preceq denotes for sub-typing). It can be seen that, according to table 1, the situations 1–3 are all type-safe. The situations 4–5 will cause the binding of \( m_1 \) to \( m_1 \) and this is type-unsafe because \( m_1 \)’s interface has the type PhDStudent that cannot provide more specific information required by the \( m_1 \)’s input type that is PartTimePhDStudent.

If we now update \( m_1 \) for a new version \( m_1 \) with the signature PhDStudent \( \rightarrow \) Staff. We can see that \( m_1 \leq_{co} m_1 \) because PhDStudent \( \leq \) PhDStudent and Lecturer \( \leq \) Staff. The situations
of 1-2 are still type-safe because they are actually the special cases of \( m_1^4 \) by regarding \( m_1^4 \) as a synonym of \( m_1^2 \).

### 2.2.2 Contravariance version binding

In determining which covariance or contravariance sub-type rules that should be applied to the checking of type-safety, only the sub-typing relationship existing between interfaces of versions is to be examined.

Contravariance version binding is the version binding considered in terms of contravariance sub-type rule[2, 5]. The contravariance rule says that a method is redefined by generalizing the input type but specializing the output type.

A general situation to be considered is that a method satisfies the contravariance sub-type rule with one of its versions or a method’s version satisfies the contravariance sub-type rule with the method. Let us consider a method \( m_2 \) and two given versions \( m_1^2 \) and \( m_2^2 \) with the signatures \( S_2 \rightarrow t_2, S_1^2 \rightarrow t_1^2 \), and \( S_2^2 \rightarrow t_2^2 \) respectively, where \( S \) denotes the input type and \( t \) denotes the output type. We assume also that \( m_2 \preceq_{ct} m_1^2 \) and \( m_2^2 \preceq_{ct} m_2 \), where \( \preceq_{ct} \) denotes the contravariance sub-typing. The type-safety of the binding of \( m_2 \) is considered in two cases:

- \( m_2^1 \) can be generalized to \( m_2 \). Therefore \( m_1^2 \) may be bound whenever \( m_2 \) is requested.
- \( m_2 \) can be generalized to \( m_2^2 \). Therefore \( m_2^2 \) may be bound whenever \( m_2 \) is requested.

In the above, we say a method is generalized means that the input type of the method is generalized to its super-type however the output type of the method is specialized to its sub-type [15].

If a version can be generalized to \( m_2 \) or \( m_2 \) can be specialized to a version, the selection of a version to be bound is arbitrary in the sense of type-safety. The following table summarizes the type-safety problems in various situations.

<table>
<thead>
<tr>
<th>Case</th>
<th>Version Binding</th>
<th>Type Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>( m_2 \circ m_2 @ m_2 )</td>
<td>safe</td>
</tr>
<tr>
<td>(2)</td>
<td>( m_2 \circ m_2 @ m_1^2 )</td>
<td>safe</td>
</tr>
<tr>
<td>(3)</td>
<td>( m_2 \circ m_1^2 @ m_1^2 )</td>
<td>compile-time error</td>
</tr>
<tr>
<td>(4)</td>
<td>( m_2 \circ m_2^2 @ m_2 )</td>
<td>safe</td>
</tr>
<tr>
<td>(5)</td>
<td>( m_2 \circ m_2^2 @ m_2^2 )</td>
<td>safe</td>
</tr>
</tbody>
</table>

Table 2: Contravariance version binding

Table 2 shows all possible situations during the covariance version binding. The interpretation of these situations is as follows. An illustration of these situations will be given in example 2.

(1) \( m_2 \circ m_2 \@ m_2 \): The method-calling to \( m_2 \) is received by \( m_2 \) and furthermore \( m_2 \) is the one actually bound.
(2) \( m_2 \circ m_2 \circ m_2^{1} \): The method-calling to \( m_2 \) is received by \( m_2 \) and \( m_2 \) is the one actually bound although \( m_2^{1} \) is the one expected to be bound. This indicates an alternative method migration that \( m_2^{1} \) can be generalized to \( m_2 \).

(3) \( m_2 \circ m_2^{1} \circ m_2^{1} \): The method-calling to \( m_2 \) is received by \( m_2^{1} \) and \( m_2^{1} \) is bound. This means that \( m_2 \) is replaced by \( m_2^{1} \), and this is not a type-safe action. A compile-time type error would be issued for this case.

(4) \( m_2 \circ m_2 \circ m_2 \): The method-calling to \( m_2 \) is received by \( m_2^{1} \) and \( m_2^{2} \) is the one actually bound. This indicates a method migration that \( m_2^{2} \) is generalized to \( m_2 \). This is a general case of type-safe method versioning where a transparent versioning mechanism is used. This situation also reflects a perfect method reuse that \( m_2 \) is replaced by \( m_2^{2} \).

(5) \( m_2 \circ m_2 \circ m_2 \): The method-calling to \( m_2 \) is received by \( m_2^{1} \) and \( m_2^{2} \) is bound. Comparing to the above situation, this also indicates a method migration that \( m_2 \) is generalized to \( m_2^{2} \). This situation happens when the method evolution causes that \( m_2 \) is not able to be bound to the method-calling sender except for \( m_2^{2} \) (e.g., \( m_2 \) is deleted).

Table 2 above does not cover other cases that correspond to the permutations of the expressions: \( m_2 \circ m_2 \circ m_2^{1} \), \( m_2 \circ m_2^{2} \circ m_2^{2} \), \( m_2 \circ m_2^{1} \circ m_2^{1} \), and \( m_2 \circ m_2^{2} \circ m_2 \) because they are not possible.

**EXAMPLE 2** Let us consider a method \( m_2 \) and its versions \( m_2^{1} \) and \( m_2^{2} \) with signatures of \( \text{PhDStudent} \to \text{Staff} \), \( \text{Student} \to \text{Lecturer} \), and \( \text{PartTimePhDStudent} \to \text{People} \) respectively. We assume that \( m_2 \preceq_{\text{ct}} m_2^{1} \) and \( m_2^{2} \preceq_{\text{ct}} m_2 \) for the facts of interfaces that \( \text{PhDStudent} \preceq \text{Student} \), \( \text{Lecturer} \preceq \text{Staff} \), \( \text{PartTimePhDStudent} \preceq \text{PhDStudent} \), and \( \text{Staff} \preceq \text{People} \). It can be seen that, according to table 2, the situations (1), (2), (4), (5) are all type-safe. The situation 3 is not type-safe because the binding of \( m_2 \) with \( m_2^{1} \) will cause the actual parameters of \( m_2 \) to be bound with the formal parameters of \( m_2^{1} \). This requires type \( \text{PhDStudent} \) to provide specific information for the type \( \text{PartTimePhDStudent} \). Obviously, \( \text{PhDStudent} \) is not a sub-type of \( \text{PartTimePhDStudent} \), a type-mismatch error should be reported in this case.

In the following, we present a general definition for the version binding.

**DEFINITION 4** (Version binding)

Given a set of method \( M = \{ m_i(S_i \to t_i) \mid \text{for } i = 1..n \} \) and the poset of corresponding version trees \( \text{Poset}_{r} = \{ \text{Poset}_{r,m_i} \mid \text{for } i = 1..n \} \), the version binding is a validation process for the type-safety of expressions and \( E = \{ (m_i \circ m_j \circ m_k^{p,q}) \mid \text{for } i,j,k = 1..n, \text{and } p,q = 1..v \} \), where \( n \) is the maximum number of methods and \( v \) is the maximum number of versions, \( m_i, m_j, m_k \in M \), \( m_p \in \text{Poset}_{r,m_i} \), \( m_q \in \text{Poset}_{r,m_k} \), such that expressions \( e \in E \) satisfy the type-safety cases in tables 1 and 2. In particular,

for \( e \in E \):

- there exists \( g \) (\( g=j \) or \( g=k \)) and \( r \) (\( r=p \) or \( r=q \)) and \( m_i \preceq_{\text{ct}} m_g^{r} \), or \( m_g^{r} \preceq_{\text{ct}} m_i \), the type-safety cases in table 1 are satisfied; or
- there exists \( h \) (\( h=j \) or \( h=k \)) and \( s \) (\( s=p \) or \( s=q \)) and \( m_i \preceq_{\text{ct}} m_h^{s} \), or \( m_h^{s} \preceq_{\text{ct}} m_i \), the type-safety cases in table 2 are satisfied.
The preceding definition says that the late binding of any versioned method involves the checking of the covariance sub-type rule as well as the contravariance sub-type rule between the method-calling sender and the receiver according to tables 1, 2 respectively.

This definition also implies a binding procedure that is carried out to determine $e \in E$. We propose a **two-level binding process**. At the first-level the traditional late binding is used, where a select-match process selects a method name and matches the type of the interfaces for the binding. At this level, the process focuses on the resolution of method overriding, overloading and migration. When this level of late binding is performed, a type-safe expression $m_i \circ m_j @ m_k$ is formed without considering the versions of $m_j$ and $m_k$. At the second level, the posets of the version trees are searched to select a version of the method that is used to replace either $m_j$ or $m_k$.

To sum up, we identify the following procedure for binding a versioned method $m$.

1. A method-calling to $m$ is issued, i.e., $m$ is called with the actual parameters.

2. The level-one binding is carried out to see if $m$ is an overridden or overloaded method. If so, a method with formal parameters will be determined for the binding. Otherwise, a correct method whose interface satisfies the covariance or contravariance sub-type rules with $m$ will be migrated for the binding.

3. The level-two binding is carried out if level-one binding is stopped with any type-mismatch errors. Level-two binding will determine a version of a method for the binding.

Note that the above procedure assumes that the underlying version mechanism is transparent, i.e. no method-calling is explicitly sent to a specific method version. When a method binding succeeds without any type-errors, no further binding on the method versions will be carried out.

The remaining question to be answered is how can we determine a method version in a version tree? That is, when a version poset is searched, what are the criteria used to pick up a method version to form a type-safe expression $e$ given by tables 1 and 2?

For a given $e_m = (m_i \circ m_j @ m_k)$, the criteria of searching a poset for type-safe versions are proposed as followings:

1. If $i = j = k$, the poset $Poset_{T_{m_i}}$ of $T_{m_i}$ will be searched for $m_i^p$, $p$ is a version number and $m_i^p \in Poset_{T_{m_i}}$, such that $m_i^p \leq_{cv} m_i$ (or $m_i \leq_{cv} m_i^p$) or $m_i^p \leq_{ct} m_i$, (or $m_i \leq_{ct} m_i^p$). Then replace $e_m$ as $e_v = (m_i \circ m_j @ m_k)$, Then check either the table 1 or 2 for the type safety.

2. If $i \neq j \neq k$ (where $i \neq k$) or $i = j \neq k$, the poset $Poset_{T_{m_i}}$ of $T_{m_j}$ will be searched for $m_j^p$, $p$ is a version number and $m_j^p \in Poset_{T_{m_j}}$, such that $m_j^p \leq_{cv} m_j$ (or $m_j \leq_{cv} m_j^p$), or $m_j^p \leq_{ct} m_j$, (or $m_j \leq_{ct} m_j^p$. Then replace $e_m$ as $e_v = (m_i \circ m_j @ m_k)$. Then check either the table 1 or 2 for the type safety. In this case, $m_k$ is regarded as irrelevant because the focused methods in the binding process are $m_i$ and $m_j$.

3. If $i \neq j \neq k$ (but $i = k$) or $i \neq j = k$, the poset $Poset_{T_{m_k}}$ of $T_{m_k}$ will be searched for $m_k^p$, $p$ is a version number and $m_k^p \in Poset_{T_{m_k}}$, such that $m_k^p \leq_{cv} m_k$ (or $m_k \leq_{cv} m_k^p$), or
Then replace $e_m$ as $e_v = (m_i \circ m_j @ m_k^p)$. Then check either the table 1 or 2 for the type safety.

The preceding criteria can be justified by the principle that if a type-mismatch error is detected in the late binding process, the only remaining step is to see if any of the method versions can be bound without type-errors.

3 Conclusion

In this paper, we have addressed the problem of type safety of methods in the context of method versioning. A two-level binding process is proposed. The first level relates to the traditional late binding plus the handling of migrated methods. The second level is the version binding that determines a correct method version to be bound with respect to the type safety. These two levels of late-binding process can be applied to a transparent-method-versioning mechanism in which the type safety is considered prior to the decision of a specific version. The significance of this proposed approach is the application of polymorphism for method evolution by allowing the versioned methods to be type-safely executed.

Acknowledgment

The author would like to thank Dr Kang Zhang for his encouragement to submit this work. The author also express his thanks to Dr Zahir Tari for some comments made on the early draft of this paper.

References


