Extending CSCM to support Interface Versioning

Hamdan Msheik, Alain Abran
Software engineering department, École de Technologie Supérieure, Université du Québec
1100 Notre-Dame Ouest, Montréal, Québec
Canada H3C 1K3
hamdan.msheik.1@ens.etsmtl.ca, aabran@ele.etsmtl.ca

Abstract-**Software component has been a main stream technology used to tackle issues such as software reuse, software quality and, software development complexity. In spite of the proliferation of component models (CORBA, .Net, JavaBeans), certain issues and limitations inherent to components are still not addressed adequately. For instance, composing software components especially those provided by different suppliers may result in faulty behavior. This behavior might be the result of incompatibilities between aging components and/or freshly released components and their respective interfaces. This paper, present an approach to tackle component interface incompatibilities via the use of a component and interface versioning scheme. This approach is designed as an extension to the Compositional Structured Component Model (CSCM), an ongoing research project. The implementation of this extension makes use of code annotations to provide interface versioning information useful in detecting interface incompatibilities.**

I. INTRODUCTION

Software development has been evolving for the past 30 to 40 years over several software development paradigms: structured, functional, and object oriented. Lately, the component based software engineering paradigm has been gaining significant attention.

This component based software engineering paradigm has been considered by Peter Maurer as a computing revolution on a par with those of stored programs and programming languages [1]. However, the idea behind software components is not new: it first appeared in a NATO conference on software engineering in the late 1960's [2].

Software components have been defined in many different ways [1, 3-6] The common characteristics among these definitions are: a) they have interfaces and interface implementations used in interconnecting with other components; b) their behavior is almost independent; and c) their form is binary so that they can be treated as black boxes. Another definition from Heineman [6] goes a little further and requires a software components to comply with a component model which defines components interactions and composition standards.

In a context of growing software functionalities, increased software complexity and ever changing requirements, component based software development has been proposed as the answer. The Software component technology addresses several issues, including: better software reuse and modularity, better quality and easier and faster development. Not addressing these issues adequately often leads to monolithic [7, 8] software applications which are less flexible, more complex, difficult to reuse and costly to develop and maintain.

Several component models (CORBA, .Net, EJBs, JavaBeans) have been developed to address the complexity of software applications, increase the potential for software reuse and enhance software distribution and interoperability. Originally, these models were introduced to address issues such as applications interoperability [9], object distribution [10], and rapid GUI (Graphical User Interface) construction. While the aforementioned component models represent significant technological improvements, they still have several limitations.

When developing software application families, considerable effort is expended on the adaptation, and customization of the functionalities of components shared by the various constituent applications. Typically, the set of useful and required functionalities provided by a particular component varies according to the particular software application context. Put differently, a number of functionalities provided by certain components are not used by their applications, compromising application integrity and security in addition to wasting memory. To tackle this limitation, the CSCM component model [11] provides a solution to the unwanted functionalities exhibited by software components.

However one important limitation of the CSCM component, as well as for the above mentioned component models, is their lack to support component interface versioning. In this paper, we present an extension to the CSCM model to address this limitation. This extension is to allows CSCM components to:

- Eliminate the problems that might arise from the use of incompatible component interfaces;
- Improves the overall quality of a software component by reducing application faults and bugs related to interface obsoleteness;
- Contribute to the enhancement of the CSCM model to provide an easier and more flexible software customization, adaptation, and reuse approach for component based software construction.

Section 2 starts by presenting background information on the CSCM model, interface definition languages and component interface versioning. Section 3 presents the CSCM model versioning scheme. Finally, a summary and a discussion of current and future works are presented in section 4.

II. Background

A. CSCM Overview

The Compositional Structured Component Model (CSCM) [11] is designed to construct software components based on selective functional composition. CSCM component functionalities are partly selected based on metadata information provided by a metadata composition descriptor instance associated with that particular component. This selective composition property provides flexibility for adapting and customizing components, as well as for facilitating software maintenance and helping to more readily achieve software reuse.

A CSCM component instance is an object with enhanced capabilities allowing selective functional composition of disjoint compositional parts (see Fig. 1). A compositional part is a method implemented independently and disjointly from a component implementation to which it is attached. We call such a part a compositional interface for being independently implemented and physically disjoint from the component's implementation.

CSCM component instances can be considered as objects since they possess and exhibit similar properties and characteristics to those of objects. CSCM components support inheritance; however, they are provided with a powerful composition and retrogression mechanism which allows CSCM components to either include or exclude compositional interfaces. This inclusion and exclusion of functionalities is done according to information provided by the component metadata composition descriptor instance.

CSCM components mechanism of composition and retrogression is based on:

- An extension to the syntax of an object oriented programming language to support compositional members.
- The use of the composition principle to selectively include the required functionalities suitable for a software application.
- The use of the delegation principle which permits the dispatching of the component methods' calls to the compositional interfaces of the component.

CSCM component's compositional interfaces are methods and not types in comparison to Java interfaces. A CSCM component compositional interface differs from CSCM ordinary methods in that they are selectable via the component metadata composition descriptor instance. In other words, the same component in two different applications might have different subsets of compositional interfaces, depending on the functional requirements needed by the hosting application.

B. CSCM component structure

A CSCM component is a software part possessing compositional interfaces, and composition descriptor which captures metadata information specifying various aspects, characteristics, dependencies and properties necessary for functional composition.

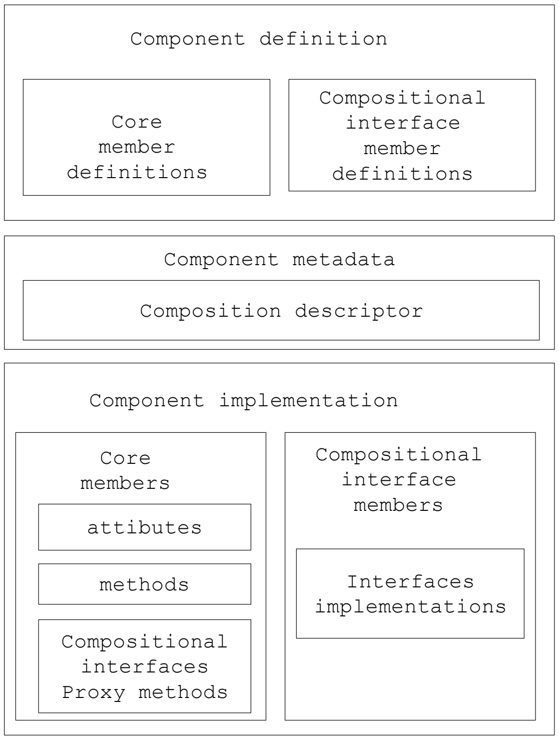
Component definition
Core member definitions
Compositional interface member definitions
Component metadata
Composition descriptor
Component implementation
Core members
attibutes
methods
Compositional interfaces Proxy methods
Compositional interface members
Interfaces implementations

Fig. 1. CSCM component structure

The structure of a CSCM component (see Fig. 1) is composed of three logical parts: a definition part, a metadata part and an implementation part.

CSCM component definition: The component definition part includes the definition of two distinct categories of members: core members and compositional interface members.

Core members are composed of method and attribute members acting as the component core for any CSCM component instance. Compositional interface members are selectively available for composition through CSCM component instances.

As illustrated in Table I, a CSCM component definition has to be defined in a file similar to the way object oriented source classes are defined. This definition defines the component "Compressor" with one core method displayUsage() and one compositional interface zip(…).

CSCM component metadata: A CSCM component composition descriptor part captures metadata information in XML format. The composition descriptor provides all necessary information on a component's members and their dependencies so that at composition-time the selection of an interface will also result in load-time selection of all compositional members on which the interface depends. A partial sample of a composition descriptor for the component shown in Table I is illustrated in Table II.

CSCM component implementation: The CSCM component implementation part contains one core implementation class for the component core members and a different implementation class for each compositional interface. Besides the implementation of core members, a component core class also provides a proxy delegate method for each compositional interface.

TABLE I
COMPONENT DEFINITION

```
import java.io.File;

@ComponentVersion (
   name = "Compressor",
   major = 1,
   minor = 2,
   micro = 1)
Component Compressor {
public String displayUsage(){…}
// compositional members
  // zip algorithm
Compositional public File zip(File f,
                       String oper){}
}
```

TABLE II
A PARTIAL ILLUSTRATION OF THE COMPOSTION DESCRIPTOR ELEMENTS FOR THE COMPONENT DEFINED IN TABLE I

```
<component name"Compressor">
  <version>
    <major>1<major>
    <major>2<minor>
    <major>1<micro>
  <version>
  </import-list>
  <core-members>
    <methods>
      <method name="displayUsage"
                return-type="String"/>
        <modifiers-list scope="public"/>
        <parameters/>
        <dependency-list/>
          <methods>…</methods>
          <attributes>…<attributes>
        </dependency-list>
      </method>
    </methods>
   </core-members>
   <compositional-members>
    <attributes/>
    <interfaces>
       <interface name="zip" selected=
           "true" return-type="File"/>
         <modifiers-list scope="public"/>
         <parameters>
          <parameter Type="File" name="f">
          <parameter Type="String"
                         name="oper">
         </parameters>
         <dependency-list/>
           <methods>
             <method name="displayUsage"
                    return-type="String"/>
             <modifiers-list
                         scope="public"/>
             <parameters/>
             </dependency-list>
         </dependency-list>
       </interface>
     </interfaces>
  </compositional-members>
</component>
```

C. *Component interfaces and interface definition languages*

Component interfaces can be considered as communication channels between components whether these components are deployed in a local or external computation environments. Component interfaces are usually specified using an IDL (Interface Definition Language). A component interface can be considered as a type which contains a set of method signatures whose implementations are provided by particular components. Current IDL languages do not allow the developer to specify interface extra-information which can be used during a component operational life.

A component interface conveys a description of what computation will be done while its implementation provides how the computation is done. Therefore, component interfaces provide flexibility and usefulness since they are not concerned of how the implementation will be done.

To overcome inter-components interoperability and development complexity problems, many definition languages have been designed to provide components with interfaces. Component model IDLs provide abstraction layers to reduce development effort as well provide reusable component intercommunication mechanisms within heterogeneous computing environments.

Currently available component models provide little or no semantic properties [12, 13]. Component interfaces express only functional aspects of a component without any consideration for aspects such as interface versioning. Nevertheless, attempts have been done to provide software component interfaces with semantics for different purposes. For instance, [12-15] describe approaches for providing component interfaces semantics and to express certain aspects such as quality of service and interface versioning.

D. *Component interface versioning*

Software components are distinguished by their use of interfaces to inter-communicate with each others. Obviously, components and interfaces are subject to modification and upgrade during their operational life cycle. As a result, modified interfaces can be a source of incompatibilities. For instance, different suppliers can provide different interface implementations (versions) for the same component interface either as updates or as fresh interface releases. These interface versions might be incompatible with a particular component version. To remedy this interface incompatibility problem, detection for such incompatibilities can be done at load-time before program execution. Eventually, integrity problems and/or data corruption can be avoided due to faulty behavior caused by interface incompatibilities.

Several tools and frameworks have been used as IDL extensions [12, 16] to annotate component interfaces. Such annotations provide metadata information useful for expressing various interface non functional aspects. To our knowledge, interface annotations have not been used yet to represent component and interface versioning information. The novelty of our approach is the use of annotation to represent component and interface versioning metadata information.

TABLE III
COMPOSITIONAL INTERFACES SKELETONS PARTIAL IMPLEMENTATION

```
  /* The code is generated by the CSCM
     Compiler */

@InterfaceVersion(
   interfaceVersion =
     @ComponentVersion (
        name = "zipClass",
        major = 3,
        minor = 3,
        micro = 1),
   compatibleComponentVersion = {
     @ComponentVersion (
        name = " zipClass",
        major = 3,
        minor = 3,
        micro = 1)}
  )
Public class zipClass {
public File zip(File f, String oper) {
  __This.displayUsage();
  return new File("testTextFile.txt");
}
}
```

III. CSCM INTERFACE VERSIONING SCHEME

A. *CSCM interfaces*

CSCM interface are object methods defined inside the component. However, their implementation is provided externally in a separate class. They can be thought about as method signatures which receive a list of parameters as input and returns back a value as output. The reason behind providing their implementation externally is to give the developer the ability to select only those interfaces required to satisfy his particular application requirements.

While their implementation is done separately outside their owner's component, CSCM component interfaces are not "types" like Java types, COM or CORBA interfaces; this is quite an important difference between CSCM interfaces and the interfaces of other component models. CSCM component interfaces are not defined using an IDL language. They are defined as ordinary methods augmented with the modifier "compositional" to differentiate them from ordinary methods. This modifier signals their presence for the CSCM compiler which handles them appropriately and generates their skeleton implementations.

B. *CSCM model interface versioning scheme*

One of the drawbacks of the CSCM model is its lack to support component and interface versioning. Therefore, we present in this paper a scheme and a specification of such versioning information.

This scheme associates each component and each of its interfaces with metadata versioning information. The software component developer is the party responsible for providing this versioning information. Ultimately, at application load time, component versioning information can be checked to verify whether the deployed interfaces are compatible with the specific component version used by the application. Eventually, if incompatible interfaces were detected, the application will be handled adequately leading to a safe program halt without risking data corruption or loss. Checking for interface or component incompatibilities can be done via extending or providing the class loader with an interface compatibility detector.

C. *CSCM versioning information implementation*

The versioning information is expressed in terms of code annotations for each component and interface. Following industrial conventions of associating three numbers with a specific version (major, minor, micro), each component has to be associated with a specific version and similarly for each interface. In addition to its specific version information, component interfaces are also associated with a set of component versions to which a particular interface is compatible with. Implementing this versioning scheme requires language constructs to express versioning information. Fortunately, Java 5.0 has been provided with such annotations constructs [17]. Table V shows CSCM component version annotations. Table VI shows CSCM interface version annotations which, in addition to expressing the interface version, expresses information on the component versions with which this interface is compatible. CSCM Component and interface version annotations are illustrated in the example component shown in Tables I, II, III and IV.

Attaching CSCM component and interface versioning annotations are the responsibility of the component and interface developer. On application startup and before the selection of compositional interfaces, a compatibility check is done automatically to detect possible interface incompatibilities. If incompatibilities are detected, the program stops execution in a predictable manner. The code responsible for interface compatibility checking is generated by the CSCM compiler and is injected inside the constructor of the component. It can be argued that this new feature of the CSCM model incurs a certain computational overhead due to checking of instance incompatibilities. However, this computational overhead can be optimized by requiring only freshly installed and upgraded CSCM components and interfaces to be checked for compatibility.

IV. DISCUSSION

In this paper, we have presented an extension to the CSCM model to address the issue of interface incompatibilities. The approach we used to handle this issue is based on the use of annotations which are now supported natively by Java 5.0. Tackling interface incompatibilities using versioning is important in particular to reduce faulty behavior and data corruption or loss when incompatible interfaces are used. Remedying this issue of interface incompatibilities is particularly significant since CSCM component interfaces are method-like and disjointly implemented. This new capability offered by CSCM component builds upon its other advantages to improve software quality and to ease and simplify software maintenance, modification and reuse.

Component models have emerged to enhance software reuse, increase application flexibility and reduce maintenance. A number of these component models lack the mechanisms to express non-functional aspects such quality of service properties and interface versioning. Several research works have provided extension to IDLs to express component non-functional aspects. For instance, fault tolerance has been addressed in [14] by extending CORBA IDL.

Similarly, in [15] a specification for an extension to an IDL has been given to express aspects such as quality of service properties. To our knowledge, little effort has been spent on the issue of interface incompatibilities. The novelty in our approach is the use of Java 5.0 annotations to provide CSCM components and interfaces with metadata versioning information to detect interface incompatibilities.

Current work in progress includes the development of the CSCM compiler with interface versioning. The implementation of CSCM compiler is targeting Java 5.0, for it supports annotations natively. Future work will explore and address distribution and interoperability through integration of the CSCM with other component model such as CORBA. Once the implementation is ready, validation with a variety of components and applications will be conducted.

TABLE IV
COMPONENT CORE PARTIAL IMPLEMENTATION

```
@ComponentVersion (
   name = "Compressor",
   major = 1,
   minor = 2,
   micro = 1)
public class Compressor {
  /* this code is generated by CSCM
     compiler */

// core method
  public String displayUsage(){
  return new String();
  }
  // compositional interfaces
  Private File zip(File f,String oper){
  return((zipClass)
      compositionalInterfaces.get(
             "zip")).zip(f, oper);
   }
 }
```

TABLE V
ILLUSTRATES A COMPONENT VERSION ANNOTATION

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@interface ComponentVersion {
String name()
int mainVersion ();
int minorVersion();
int microVersion();
}
```

TABLE VI
ILLUSTRATES AN INTERFACE VERSION ANNOTATION

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@interface InterfaceVersion {
 ComponentVersion interfaceVersion();
 ComponentVersion []
            compatibleComponentVersion();
}
```

REFERENCES

[1] P. Maurer, Component-Level Programming: Pearson Education Inc., 2003.
[2] M. D. McIlroy, "Mass Produced Software Components," presented at NATO Software Engineering Conference, 1968.
[3] C. Szyperski, "Emerging component software technologies--a strategic comparison," Software--Concepts & tools, pp. 2-10, 1998.
[4] D. E. C. Microsoft Corporation, "The Component Object Model Specification." http://www.microsoft.com/com/resources/comdocs.asp, 1995.
[5] I. Jacobson, "Component-based Development with UML," 1998.
[6] T. Heineman, G. and T. Council, W., "Component-Based Software Engineering," Addison-Wesely, 2001.
[7] J. Li, "A Survey on Microsoft Component-based Programming Technologies," Concordia University, Montreal 1999.
[8] L. Vanhelsuwe, Mastering JavaBeans: Sybec Inc., 1997.
[9] C. Exton, D. Watkins, and D. Thompson, "Comparisons between CORBA IDL &amp; COM/DCOM MIDL: interfaces for distributed computing," presented at Technology of Object-Oriented Languages and Systems, 1997. TOOLS 25, Proceedings, 1997.
[10] J. Ongg, "An Architectural Comparison of Distributed Object Technologies," MIT june 1997.
[11] H. Msheik, A. Abran, and E. Lefebvre, "Compositional structured component model: handling selective functional composition," presented at Euromicro Conference, 2004. Proceedings. 30th, 2004.
[12] D. Watkins and D. Thompson, "Adding semantics to interface definition languages," presented at Software Engineering Conference, 1998. Proceedings. 1998 Australian, 1998.
[13] D. C.Schmidt, D. L. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," IEEE Computer Communications Journal, vol. 21, pp. 294-324, 1998.
[14] C. Exton, "Distributed fault tolerance specification through the use of interface definitions," presented at Technology of Object-Oriented Languages, 1997. TOOLS 24. Proceedings, 1997.
[15] H.-A. Jacobsen and B. J. Kramer, "Modeling interface definition language extensions," presented at Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings. 37th International Conference on, 2000.
[16] D. Watkins, "Using interface definition languages to support path expressions and programming by contract," presented at Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings, 1998.
[17] Q. H. Mahmoud, "The All-New Java 2 Platform, Standard Edition (J2SE) 5.0 Platform," [Online]: http://java.sun.com/developer/technicalArticles/releases/j2se15langfeat/, 2004.