XCompose: An XML-Based Component Composition Framework

Position Paper

Naiyana Tansalarak and Kajal T. Claypool
Department of Computer Science
University of Massachusetts - Lowell
Lowell, MA 01854 USA
Tel: +1-978-934-3620, Fax: +1-978-934-3551
{ntansala|kajal}@cs.uml.edu

Abstract

With increasing number of components now available on the market, research and industry emphasis has shifted from the development of component models to the development of languages and other techniques to enable the composition of pre-fabricated components. We believe that frameworks for composition of components must be *flexible*, *extensible*, *re-usable* and offer some guarantees on the *correctness* of the composition. We present in this position statement a unique framework that is based on the simple hypothesis that complex component compositions can always be broken down into a sequence of simple composition operators. Based on this hypothesis, we define a set of primitive composition operators, show how primitives composition operators can be effectively combined to formulate large, more complex component composition. In this paper, we discuss the requirements for a framework that is based on the paradigm "*applications = components + composition language*" where "*composition language = composition operators + glue logic*". In particular we examine the requirements for the composition operators and the language. We also present an overview of XCompose, an XML-based component composition framework, currently underway at UMass Lowell.

Keywords: Component Composition, Composition Lanaguage, Composition Operators

1 Introduction

Component-based software engineering (CBSE) has become recognized as the enabling technology for the on-time development of high-quality and high-reliability systems using a set of well-conceived and pre-fabricated software components [10, 11, 16]. There are two key issues when considering component based software engineering: (1) the specification and implementation of components and (2) the composition of components into composite components or applications. Much of the research thus far has focused on developing frameworks for the specification and implementation of components, resulting in a rich and diverse set of component models (JavaBeans [14], CORBA [17], COM [3] etc.). More recently, research has progressed towards the development of composition approaches to enable the construction of large systems via component composition.

Current research on composition languages has focused on both the re-use of existing object-oriented languages and the development of new scripting languages to enable high-level component composition. Although the reuse of object-oriented languages holds a certain allure, and they are well-suited for implementing software components, they fail to shine in the construction of component-based applications. This is largely due to the fact that object-oriented design tends to obscure a component-based architecture [1]. More recent work has focused on the development of a special purpose composition language such as Piccola [1] that embodies the paradigm of "*applications = components + scripts*". Piccola models components and composition abstractions by means of a unifying foundation of com-

municating concurrent agents. Bean Markup Language (BML) [18] and Component Markup Language (CoML) [2] based on XML, are other examples of a scripting language that have been developed to enable component composition in a platform independent manner. Butler et al. [5] take an orthogonal approach and provide a set of *composition operators* to define object interaction at the granularity of a method.

In our previous work, we have developed a framework, SERF, that provides flexibility, extensibility and re-usability in the context of schema evolution for object-oriented database systems [7]. The SERF framework was targeted to address the limitation of other schema evolution approaches most of which provided a fixed taxonomy of schema evolution operations. The goal of the SERF framework was to allow users to perform a wide range of complex user-defined schema transformations *flexibly*, *easily* and *correctly*. The SERF approach is based on the hypothesis that complex schema evolution transformations can be broken down into a sequence of basic evolution primitives, where each basic primitive is a correctness-preserving atomic operation with fixed semantics. We use a standard query language, OQL [6], based on the ODMG [6] object model, to effectively combine these primitives and to be able to perform arbitrary transformations on objects within a complex schema operation. This query language served as the transformation glue logic.

There are many parallels that can be drawn between component composition and the specification of complex schema transformations. In this position statement, we investigate the requirements and the feasibility of developing an XML-based component composition framework, namely `XCompose`, that is based on the paradigm "*applications = components + composition language*" where "*composition language = composition operators + glue logic*". We have four primary goals for our `XCompose`. The `XCompose` should be *flexible* - in that the users should be able to compose components in a "plug-and-play" manner; *extensible* - in that the users should be able to tailor existing compositions to fit their application domain needs; *re-usable* - in that the compositions are made available to all users as a resource that can then be re-used; and *correct* - in that the users must have some system-level guarantees on the correctness of the component composition.

In our initial investigations of this work, we focus on *connection-oriented* and *aggregation-based* compositions, wherein we can describe how components are *plugged* together as well as describe the aggregation of components to present a higher-level component. Our approach, a la SERF, is based on the hypotheses that complex component compositions can be broken down into a sequence of primitive composition operators glued together by a simple language. Thus, there are three essential ingredients for achieving a flexible, extensible, re-usable, and correct component composition framework, namely (1) a set of well-defined, primitive *composition operators* that are correctness preserving. We say that an operator is correctness preserving if the contracts [13] (pre- and post-conditions) for the components participating in the composition are not violated; (2) a simple and easy to use language that provides the glue logic for combining together the primitive composition operators to enable the definition of larger, more complex component compositions; and (3) a mechanism to capture and re-use the common composition patterns in a composition environment.

In the rest of the paper, we examine composition operators in Section 2 and composition patterns and templates in Section 3. We then present an overview of our XML-based component composition framework that we are currently working on in Section 4. Finally, we conclude in Section 5.

2 Composition Operators

Composition operators, the core ingredient of the our composition framework, represent the building blocks on the basis of which more complex compositions can be defined. In this section we now discuss the requirements of a composition operator and present a possible set of composition operators.

2.1 Requirements for Composition Operators

A composition operator provides the basic manipulation of one or more components, and produces a *composite* component or an application as output. A composition operator must, in general, be able to

compose and manipulate every aspect of an existing component. This implies that the operators must provide composition both at the lowest granularity of methods of a single class within the component, as well as at the level of classes and components. However, the diversity of feasible compositions increases at the higher levels resulting in possibly infinite number of class and component combinations. To account for this growing need for flexibility in composition semantics, we break down the composition into two main categories: *primitive composition operators* and *composition patterns*. Primitive composition operators reflect the basic, most primitive composition semantics such that they cannot be broken down any further. Composition patterns, on the other hand, reflect more complex compositions, typically at the level of a class or a component such that their semantics can be expressed by a combination of the primitive composition operators. We discuss composition patterns further in Section 3.1. Here we now give the set of requirements that must be met by the primitive composition operators. A primitive composition operator must provide:

- *minimal semantics*: As the eventual goal is to enable flexible composition by combining primitive composition operators, it is essential that the semantics of the primitive composition operators be both simple and minimal. Minimal semantics imply that the semantics of the primitive composition operator cannot be expressed by any combination of the other primitive composition operators;

- *complete*: To enable full flexibility in the complex composition, it is essential that the taxonomy of primitive operators be complete. That is, primitive composition operators should be defined to express all possible compositions at the lowest granularity, i.e., at the method level;

- *correct*: This is a key requirement. Each primitive composition operator must guarantee that if the source component(s) are valid and correct, then the complex composition resulting from the application of the composition operators will also be valid and correct. As a first step, we must ensure that the primitive operators always result in correct and valid compositions.

2.2 Primitive Composition Operators

Butler [5] et al. have defined a set of operators for combining object interactions at the granularity of a method. Based on this set of operators, we now define a set of *composition operators* to enable composition not only for methods, but also for classes and components as we believe that these are the *primitive composition operators* on the basis of which other *complex composition* at both the class level and the component level can be defined. While these operators do not necessarily represent a complete set, they do provide a good first start for component composability. We also do not consider attributes per say as they can be manipulated via their accessors methods and we assume that *basic manipulation operations* for the addition, renaming and deletion of attributes, methods, and classes are available.

We now define five composition operators, namely *conjunction*, *sequence*, *choice*, *pipe* and *loop* to enable different compositions of methods from one or more classes. The *conjunction* operator, represented as $\mathtt{m}_i \wedge \mathtt{m}_j$, denotes the execution of the two methods $\mathtt{m}_i$ and $\mathtt{m}_j$ simultaneously wherein the initial state of the system, env, must satisfy the pre-conditions of both methods $\mathtt{m}_i$ and $\mathtt{m}_j$. The post-conditions of the composition method $\mathtt{m}_i \wedge \mathtt{m}_j$ are the post-conditions of the individual methods $\mathtt{m}_i$ and $\mathtt{m}_j$.

The *sequence* operator, represented as $\mathtt{m}_i;\mathtt{m}_j$, denotes the execution of the two methods $\mathtt{m}_i$ and $\mathtt{m}_j$ in sequence, wherein env must satisfy the pre-condition of $\mathtt{m}_i$, and the post-condition of $\mathtt{m}_i$ must satisfy the pre-condition of $\mathtt{m}_j$. Moreover, the post-condition of the composition method $\mathtt{m}_i;\mathtt{m}_j$ is given simply as the post-condition of the method $\mathtt{m}_j$.

The *choice* operator, represented as $\mathtt{m}_i \vee \mathtt{m}_j$, denotes that the composition method consists of the semantics of either the method $\mathtt{m}_i$ or the method $\mathtt{m}_j$ (not both). The pre- and post-conditions for this composition method are the pre- and post-conditions of the chosen method $\mathtt{m}_i$ or $\mathtt{m}_j$ (not both).

The *pipe* operator, represented as $\mathtt{m}_i \mid \mathtt{m}_j$, denotes the execution of the two methods $\mathtt{m}_i$ and $\mathtt{m}_j$ in sequence, wherein the output of the method $\mathtt{m}_i$ is the input of the method $\mathtt{m}_j$. The env must satisfy the pre-condition of $\mathtt{m}_i$, and the post-condition of $\mathtt{m}_i$ must

satisfy the pre-condition of m_j. Moreover, the post-condition of the composition method $m_i;m_j$ is given simply as the post-condition of the method m_j.

The *loop* operator, represented as m_i^*, denotes the repeated consecutive execution of the method m_i. The pre- and post-conditions of the composition method in this case are the pre- and post-conditions of n^{th} iteration of the method m_i.

3 Composition Patterns and Templates

Based on the primitive composition operators, we now show how more complex composition, termed *composition patterns*, can be created. We then look at the re-usability aspect of these composition patterns. Re-usable composition patterns are termed *composition pattern templates* or just *templates* for short. In this section we first define composition patterns and templates, and then provide a discussion, based on examples, on the requirements of a composition language.

3.1 Composition Patterns

Composition at the higher levels, that is at the level of a class or a component, can be defined via a combination of the different primitive composition operators and basic manipulation operations. Literature [12, 4] cites many complex compositions albeit in the context of object-oriented systems. For example, Lerner [12] discuss six complex operations including *inline*, *merge* and *split*. Inline is defined as moving of all attributes and methods from a referred class to the main parent class.

Figure 1 gives a pictorial representation of an *inline* composition of the two classes `Person` and `Address` to form a new class `PA`. Here the class `PA` contains the attribute `name`, and the methods `setName()` and `getName()` from the class `Person`, and all of the attributes and methods of the class `Address`. Additionally, the class `PA` also contains two new methods `getAddress ()`, and `getCityOrZip ()` wherein the `getAddress()` method returns the concatenation of the `street`, `city`, `state` and `zip` and the `getCityOrZip()` returns either the

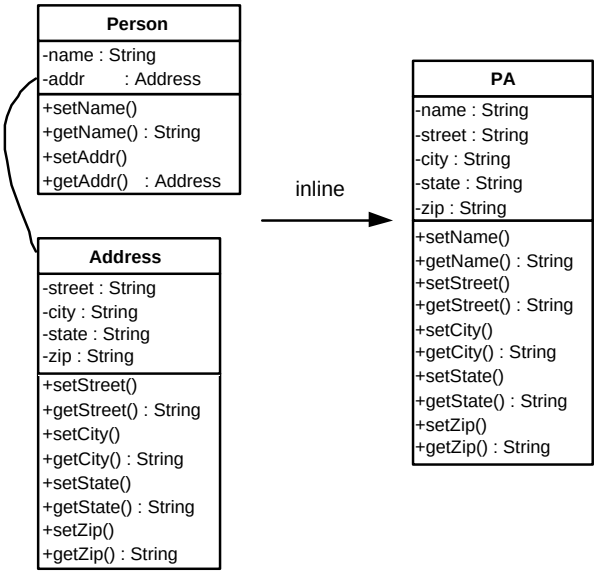

Figure 1: Pictorial Representation of an Inline Composition.

`city` or the `zip`. We term this *extended-inline* composition. This complex composition of the new class `PA` can be broken down into a sequence of primitive composition operators and basic manipulation operations. For example, the addition of the attributes and the methods from the classes `Person` and `Address` can be accomplished by the basic manipulation operations such as `add_attribute` and `add_method`. The composition of the methods `getAddress()` and `getCityOrZip()` can be given as follows:

- `getAddress()` $\leftarrow$ (`getStreet()` ; `getCity()`; `getState()`; `getZip()`).

  The method `getAddress()` is implemented as sequential executions of the four methods `getStreet()`, `getCity()`, `getState()` and `getZip()`. We assume here that the outputs of these four methods are concatenated together and returned as one string.

- `getCityOrZip()` $\leftarrow$ (`getCity()` $\vee$ `getZip()`).

  The method `getCityOrZip()` is implemented as a choice between the two methods `getCity()` and `getZip()`.

Figure 2 gives a fragment of the composition pat-

tern for this extended-inline composition. This composition pattern is a sequence of primitive composition operators and basic manipulation operations glued together by a simple pseudo language. Section 3.3 provides more discussion on the requirements for such a language.

```
create_class (PA);

add_attribute (PA, name, private, String, "");
add_attribute (PA, street, private, String, "");
....
add_method (PA, setName(), public, void, <String>);
add_method (PA, getName(), public, String, <>);
add_method (PA, setStreet(), public, void, <String>);
add_method (PA, getStreet(), public, String, <>);
...
add_method (PA, getAddress(){getAddress() ←
    (getStreet() ; getCity() ; getState() ; getZip())},
    public, String, <>);
add_method (PA, getCityOrZip(){getCityOrZip() ←
    (getCity() ∨ getZip())}, public, String, <>);
```

Figure 2: The `Extended-Inline` Composition Pattern Defined Using the Primitive Composition Operators and the Basic Manipulation Operations.

3.2 Composition Pattern Templates

Composition patterns such as the one shown in Figure 2 are complex composition operations that are *written once and used once*. These patterns are bound to the classes for which they are written, and do not offer much in the way of *re-use*. However, there may be many composition patterns such as *inline* of one class into another, *merge* of two classes, *concat* of two classes, or *diff* of two classes, that could potentially be re-used for different classes. Such composition patterns should ideally be provided as a "plug-and-play" resource for the user much as the combination of primitive composition operators. Thus, one of the desired features of a composition framework is to provide re-usability of not only the components, but also of these composition patterns. To facilitate this pattern re-usability, a composition pattern must necessarily be generalized. Thus, we must now modify the composition pattern such that it can be applied not just to the component(s) for which it was initially written but also for any component(s) in general. In order to facilitate this, we must remove any references to the particular component(s); and allow the composition pattern to have a name, a set of input parameters, and variables to allow the binding of the parameters.

We now introduce the notion of *composition pattern templates* or simply *templates*. Composition pattern templates are, thus, named, parameterized, and generalized composition patterns. Figure 3 gives an example of an extended-inline composition pattern template based on the extended-inline composition pattern in Figure 2. Here we have given the composition pattern a name `exInlineTemplate`. The `exInlineTemplate` has five input parameters, namely `sourceClass`, `inlineClass`, `newClassName`, `addressMethod`, and `cityZipMethod`. The input parameter `sourceClass` represents the main class (`Person`); the input parameter `inlineClass` represents the class that is to be inlined (`Address`); the input parameter `newClassName` gives the name of the new class (`PA`); the input parameter `addressMethod` contains the names of the methods that must be used to formulate the `getAddress()` method; and the input parameter `cityZipMethod` contains the set of method names that must be used to formulate the `getCityOrZip()` method. Additional input parameters can be added to the `exInlineTemplate` to allow users to specify the names of the methods `getAddress()` and `getCityOrZip()`. In this case we have chosen system defined names for these methods. The next step is the generalization of the `extended-inline` pattern. Clearly, it is not possible to know, in a general manner, the attributes and methods of any given input class. Thus, the statements **(1)** and **(2)** in Figure 3 show how the composition patterns can now be generalized via the use of meta-data. All variables in the template are preceded by a $ sign.

3.3 Requirements of a Composition Language

Based on the `extended-inline` example presented in Figure 2 and Figure 3, we now examine the requirements of a composition language. This composition language provides the glue logic to enable the specification of compositions templates, that is it allows users to combine the primitive compo-

exInlineTemplate(sourceClass, inlineClass, newClassName, addressMethod[], cityZipMethod[])
{
create_class ($newClassName);

forall attributes a_i_ ∈ $sourceClass do: **...(1)**
  if (a_i_.type ≠ $inlineClass) do:
    add_attribute ($newClassName, a_i_, a_i_.scope(), a_i_.type(), a_i_.initialValue());
forall attributes a_i_ ∈ $inlineClass do:
  add_attribute ($newClassName, a_i_, a_i_.scope(), a_i_.type(), a_i_.initialValue());
forall methods m_i_ ∈ $sourceClass do: **...(2)**
  if (m_i_.returnType ≠ $inlineClass and $inlineClass ∉ m_i_.paraList) do:
    add_method ($newClassName, m_i_, m_i_.scope(), m_i_.returnType(), m_i_.paraList);
forall methods m_i_ ∈ $inlineClass do:
  add_method ($newClassName, m_i_, m_i_.scope(), m_i_.returnType(), m_i_.paraList);

define $tmp1 = "$A; $B; $C";
add_method ($newClassName, getAddress(){getAddress() ← ($tmp1 ← $addressMethod[])}, $tmp1.scope(), $tmp1.returnType, $tmp1.paraList()) ;

define $tmp2 = "$A ∨ $B";
add_method ($newClassName, getCityOrZip(){getCityOrZip() ← ($tmp2 ← $cityZipMethod[])}, $tmp2.scope(), $tmp2.returnType, $tmp2.paraList()) ;

Figure 3: The `Extended-Inline` Composition Pattern Template.

sition operators to formulate arbitrarily composition templates based on some underlying components. A composition language must thus be expressive to enable different combinations and semantics for a complex composition. The most rudimentary requirements for such a language are: *iteration*, that is the ability to iterate over the meta data - the attributes and methods of the class, and the classes in a component; *conditional statements*, that is to allow for conditional statements (`if then else` construct); *existential and universal quantification*, that is to allow checks for some or all entities of a set; and *type system*, that it must be strongly typed. In addition, the language must allow variable binding, path manipulation, and must be portable, that is platform independent. Furthermore, it can be seen that such requirements are able to support the requirements of a composition language defined in [15, 2] such as composition code reuse and extensibility.

To translate a composition pattern template to a composition pattern, we must necessarily have access to the meta-data that describes, for example, the interface of the component, classes in a component, and the attributes and methods of a particular class etc. Thus, to enable re-usable composition pattern templates a composition language must meet the additional requirements such as (1) allow the notion of a method with parameters; (2) bind variables to input parameters; and (3) provide access to the component meta-data.

4 Overview

We are currently in the process of developing an XML-based component composition framework, namely `XCompose`. In this section, we briefly outline the framework and describe its key features.

Figure 4 gives an architectural overview of our proposed `XCompose`. `XCompose` is being developed as a thin layer on top of existing component frameworks such as JavaBeans [14], CORBA [17], and COM [3]. In the figure the top half represents the `XCompose`, while the bottom half represents the underlying component frameworks.

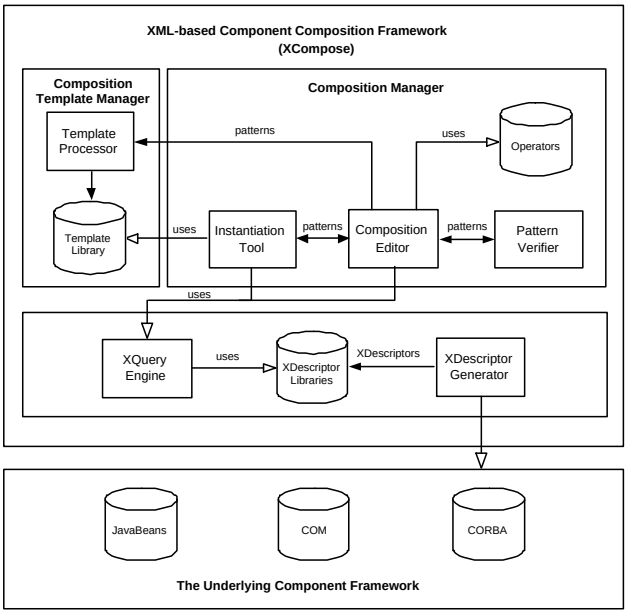

Figure 4: Architecture of the XML-Based Component Composition Framework- `XCompose`.

Given the wide acceptance of the XML model [8] and its suitability to provide the portability of compositions as well as the component model indepen-

```
<?xml version="1.0" encoding="UTF-8"?>
<class name= "PA">
  <composition operator = "inline">
    <classes>
      <class id="person" name="Person"> </class>
      <class id="address" name="Address"> </class>
    </classes>
    <data>
      <attribute id="name" refID="person.name">
      </attribute>
      <attribute id="street" refID="person.addr.name">
      </attribute>
      .....
    </data>
    <behaviors>
      <method id="setName" refID="person.setName">
      </method>
      <method id="getName" refID="person.getName">
      </method>
      <method id="setStreet" refID="person.addr.setStreet">
      </method>
      <method id="getStreet" refID="person.addr.getStreet">
      </method>
      .....
      <method id="getAddress" refID="person.addr.getStreet ;
        person.addr.getCity ; person.addr.getState ;
        person.addr.getZip"> </method>
      <method id="getCityOrZip" refID="person.addr.setCity
        \/ person.addr.setZip"> </method>
    </behaviors>
  </composition>
</class>
```

Figure 5: The XDescriptor for the class PA.

dence, we use XML as the underlying model for the XCompose system. For existing components, XCompose thus generates the XML documents to represent both the architectural layout of the component and the details on the components public interface and its internal classes. These XML documents are termed *XDescriptors*. XCompose also generates XDescriptors for every composition that is created by the application of any composition operator or pattern. Figure 5 gives a fragment example XDescriptor for the class PA generated by the application of the extended-inline composition pattern given in Figure 2.

Based both on the requirements that we have laid out for a composition language as well as the use of XML to describe XDescriptors, we have chosen XQuery [9] as our choice for the component composition language. XQuery [9] is a declarative query language that not only provides iteration, conditional statements, existential and universal quantification, but also allows for the extraction of information from the XDescriptors and the access of meta-data represented in XML format. Figure 6 depicts the extended-inline composition template pattern now re-written using XQuery.

The core of the XCompose framework is the *Composition Manager* that provides a fixed set of primitive composition operators and the ability to combine these operators via a composition language into composition patterns. Here, the *Composition Editor* takes as input one or more XDescriptors and composition patterns and applies with a set of operators to produce as output a composition pattern whose specification is given in term of XDescriptor. The instantiation of composition templates is supported via the *Instantiation Tool* that essentially checks and retrieves the meta-data for the specified components. A *Pattern Verifier* ensures the correctness of a composition pattern via validation of the specified contracts. A *Deployment Engine* handles the translation and the subsequent generation of a new complex component based on the newly generated XDescriptor.

Composition patterns can be generalized and saved as *composition pattern templates* in the *Pattern Template Library*. The Pattern template library offers reuse to the users by allowing them to browse and select existing composition pattern templates, to instantiate the templates for a given parameter list, and then finally executing them.

5 Conclusion

In this paper, we propose the *XML-based Component Composition framework* (XCompose) to provide *flexibility*, *extensibility*, *re-usability*, and *correctness* of composition. Our approach is based on the hypothesis that a set of primitive composition operators can be combined to produce arbitrarily complex compositions, *composition patterns*, thereby providing flexibility and extensibility. Moreover, to increase the re-usability of the system, we also introduce the notion of composition pattern templates, wherein the composition patterns can be saved in a template library for later re-use. A key advantage of our work is the provable correctness of the compositions. We are currently in the process of developing the XCompose as a thin-layer of functionality

```
begin template exInline (sourceClass, inlineClass, newClass, addressMethod[], cityZipMethod[])
{
    define localAttrs ($cName) as
                FOR $attr IN document("dictionary.xml")//metaData
                WHERE $attr/metaClass = $cName
                RETURN $attr//localAttr

    define localMets ($cName) as
                FOR $met IN document("dictionary.xml")//metaData
                WHERE $met/metaClass = $cName
                RETURN $attr//localMet

    define refAttrName as
                FOR $ref IN document("dictionary.xml")//metaData
                WHERE $ref/metaClass = $cName1
                and $ref//attrType = $cName2
                RETURN $ref//attrName

    write (<?xml version="1.0" encoding="UTF-8"?>);
    write (<class name = $newClass >);
    write (<composition operator = "inline" >);
    write (<classes>);
    write (<class id = x1 name = $sourceClass>);
    write (<class id = x2 name = $inlineClass>);
    write (</classes>);
    write (<data>);

    Let $ref = refAttrName ($sourceClass, $inlineClass);

    forall attrs in localAttrs($sourceClass) and attrs/attrType <> $inlineClass
        write (<attribute id = attrs//attrName refID = x1.attrs//attrName>) ;
        write (</attribute>);

    forall attrs in localAttrs($inlineClass)
        write (<attribute id = attrs//attrName refID = x1.$ref.attrs//attrName>) ;
        write (</attribute>);

    write (</data>);
    write (<behavior>);
    forall mets in localMets ($sourceClass) and mets/referType <> $inlineClass
        write (<method id = mets//metName refID = x1.mets//metName>) ;
        write (</method>);

    forall mets in localMets ($inlineClass)
        write (<method id = mets//metName refID = x1.$ref.mets//metName>) ;
        write (</method>);

    define $tmp1 = "$A; $B; $C";
    write (<method id = getAddress refID = ($tmp1 ← $addressMethod) >) ;
    write (</method>);

    define $tmp2 = "$A ∨ $B";
    write (<method id = getCityOrZip refID = ($tmp2 ← $cityZipMethod) >) ;
    write (</method>);

    write (</behavior>);
    write (</composition>);
    write (</class>);
}
end template
```

Figure 6: The XQuery-Based `extended-inline` Composition Pattern Template.

on top of existing component models. We use XML as our middle-layer model to express the descriptors of the components, and make use of the XQuery [9] language to express the more complex composition,

that is the composition patterns.

Future work needs to focus on the depolyment engine which is the key part in order to verify that our approach is applicable for the component composition environment.

In this position statement, we present a general outline of our framework. However, many questions must still be addressed to take this framework to reality. Some of the questions that we are currently looking to address are:

- What is the complete set of primitive and complex operators? And what are their semantics?
- What extensions are needed to extend our framework to make it platform independent and capable of handling a heterogeneous set of component models ?
- Are XML and XQuery the right model and language for this framework ?
- What are the requirements and constraints to achieve the four primary goals of this framework ? Are these goals sufficient ?

References

[1] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola – a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press, 2001.

[2] Dietrich Birngruber. Coml: Yet another, but simple component composition language. In *Workshop on Composition Language*, 2001.

[3] Don Box. *Essential COM*. Addison-Wesley Publishing Company, 1998.

[4] Philippe Breche. Advanced principles for changing schemas of object databases. In *Conference on Advanced Information Systems Engineering*, pages 476–495, 1996.

[5] S. Butler and R. Duke. Defining composition operators for object interaction. *Object Oriented Systems*, 5(1):1–16, 1998.

[6] R.G.G Cattell and et al. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Inc., 1997.

[7] K.T. Claypool, J. Jin, and E.A. Rundensteiner. SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. In *Conference on Information and Knowledge Management*, pages 314–321, November 1998.

[8] World Wide Web Consortium. Extensible markup language (xml). http://www.w3.org/XML.

[9] P. Fankhauser, M. Fernandez, A. Malhotra, M. a Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. http://www.w3c.org/TR/2001/query-semantics, November 2001.

[10] George T. Heineman and William T. Councill. *Component-based Software Engineering*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2001.

[11] Gary T. Leavens and Murali Sitaraman. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[12] Barbara Staudt Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, 2000.

[13] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[14] Sun Microsystems. Javabeans, 1997.

[15] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Langages for Concurrent Systems*, volume 924, pages 147–161. Springer-Verlag, 1995.

[16] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, 1995.

[17] J. Siegel. *CORBA: Fundamentals and Programming for the 21st century*. John Wiley, New York, 1996.

[18] Sanjiva Weerawarana, Francisco Curbera, Mattew J. Duftler, David A. Epstein, and Joseph Kesselman. Bean markup language: A composition language for javabeans components. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology System (COOTS 2001)*, pages 173–187, January 2001.