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 must at the same time provide guarantees on the correctness of the composition. In this paper, we present an XML-based component composition framework, namely XCompose, that is based on (1) the simple hypothesis that complex component compositions can always be broken down into a sequence of primitive composition operators; and (2) the paradigm “applications = components + composition language” where “composition language = operators + glue logic”. Based on this hypothesis and paradigm, we define a set of operators and show how operators can be effectively combined to formulate arbitrary component compositions via composition pattern, thereby enabling flexibility and extensibility. In this paper, we take a step further and introduce the notion of composition templates that facilitate the re-usability of component compositions. To address the problem of composition correctness, we introduce contracts as a mechanism to specify pre- and post-conditions for both individual compositions as well as for entire applications. Lastly, we present an architectural overview of XCompose, our XML-based composition framework.

Keywords: Component Composition, Method Composition, Composition Pattern, Composition Pattern Template, Composition Language, 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 [11, 13, 17]. There are two key issues that must be addressed 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 such as JavaBeans [15], CORBA [18], COM [3]. More recently, research has progressed towards the development of 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 special purpose composition languages 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 communicating concurrent agents. Bean Markup Language (BML) [20] and Component Markup Language (CoML) [2] based on XML, are other examples of scripting languages that have been developed to enable component composition in a platform independent manner. Butler et al. [4] take an orthogonal approach and provide a set of composition operators to define object interaction at the granularity of a method.

From an application developer’s perspective, while these approaches allow the composition of components in a flexible and extensible manner, they do not provide re-usability of existing or previously developed compositions. Nor do these approaches address the critical aspect of composition correctness. In our work, we now define four general goals that must be satisfied by any composition framework, namely flexibility - to enable users to compose components in a "plug-and-play" manner; extensibility - to allow users to tailor existing compositions to fit their current needs; re-usability - to make compositions available to all users as a resource that can then be re-used; and correctness - to provide users system-level guarantees on the correctness of their component compositions. To address these requirements for a component composition framework, in this paper, we present an XML-based component composition framework, XCompose.

In our 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 is based on (1) the hypothesis that complex component compositions can be broken down into a sequence of primitive composition operators glued together by a simple language; and (2) the paradigm “application = component + composition language” where “composition language = operators + glue logic”. Thus, there are four 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 [14] (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, termed composition patterns; (3) a mechanism to capture and re-use the common composition patterns in a composition environment, termed composition templates; and (4) a mechanism to specify and subsequently verify the correctness of the component composition via composition contracts.

Roadmap: The rest of paper is organized as follows. We introduce the primitive and complex oper-
ators in Section 2 and 3 respectively. Section 4 illustrates the flexibility, extensibility and the re-usability of our by showing how complex component compositions can be specified via a combination of the primitive composition operators as well as other pre-defined component compositions. Section 5 outlines an initial strategy for the verification of composition correctness. Section 6 gives a general overview of the XCompose framework. Section 7 presents related work and we conclude in Section 8.

2 Primitive Operators

Composition operators, the core ingredient of the our composition framework, represent the building blocks on the basis of which the arbitrarily complex component compositions can be defined. However, component composition requires both manipulation of the underlying classes that form the component and the establishment of connections between two or more components. We thus distinguish pure composition operators from manipulation operators, where composition operators provide semantics to establish relationships between components and the manipulation operators provide the basic operators for the modification of the component schema\(^1\). We use the term composition to refer to both composition and manipulation operators when specific distinction is not necessary.

In this section, we first outline the general requirements for the primitive composition operators, and then go on to describe the set of primitive manipulation and composition operators that we have defined as part of our work.

2.1 Requirements for Composition Operators

Primitive composition operators reflect the basic, most elementary composition semantics such that they cannot be broken down any further. A primitive composition operator must thus satisfy the following requirements.

- **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 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 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.

\(^1\)We use the term “component schema” to imply the set of classes, properties, methods and events that form the component.
2.2 Primitive Manipulation Operators

In all component models, a component is typically built on top of classes [9, 12] where a class comprises of a set of attributes and methods. Thus, manipulation of attributes and methods, such as the addition of an attribute or the invocation of a method, are an integral part of the composition of components. We therefore now define four primitive manipulation operators, namely \textit{addAttribute, addMethod, invokeMethod} and \textit{addComponent} operators.

The \textit{addAttribute} operator, denoted as \texttt{addAttribute}(l, m, t, c, iv), creates a new attribute with a label \(l\), modifier \(m\), type \(t\), cardinality \(c\), and an initial value \(iv\), and adds it to the specified class \(C\). The operation succeeds if class \(C\) does not already contain an attribute with label \(l\).

The \textit{addMethod} operator, denoted as \texttt{addMethod}(l, m, ret, pList, pre, post, def), creates a new method and adds it to the specified class \(C\). Here \(l\) denotes the name of the method, \(m\) the modifier, \(ret\) the return type, \(pList\) the list of input parameters, \(pre\) and \(post\) the pre and post-conditions of the method, and \(def\) the definition of the method written in the native programming language supported by the underlying component model. The operation succeeds if class \(C\) does not already contain a method that has the same signature as this newly added method. The signature is represented by the label, return type, and the parameter list.

The \textit{invokeMethod} operator, \texttt{invokeMethod}(i, l, r, pList), invokes a method in the specified component or class instance. Here \(i\) refers to a specific instance of the component or class \(C\), \(l\) the label of the method, \(r\) the output of the method, and \(pList\) the set of expected parameters for the method. The operation succeeds if the component or class instance \(i\) contains a method with label \(l\), and \(r\) and \(pList\) conform to the types specified in the method signature.

The \textit{addComponent} operator, \texttt{addComponent}(c, i), creates the component instance \(i\) from the component \(c\) and adds it to the specified class \(C\). The operation succeeds if class \(C\) does not already contain the instance \(i\).

2.3 Primitive Composition Operators

Butler [4] 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 \textit{composition operators} to enable composition of not only methods, but also of classes and components as we believe that these are the \textit{primitive composition operators} on the basis of which other \textit{complex composition operators} at both the class level and the component level can be defined. We now define five composition operators, namely \textit{conjunction, sequence, choice, pipe, and loop} to enable composition of methods from one or more classes.

The \textit{conjunction} operator, represented as \(m_i \land m_j\), denotes the execution of the two methods \(m_i\) and \(m_j\) simultaneously wherein the initial state of the system, \texttt{env}, must satisfy the pre-conditions of both methods \(m_i\) and \(m_j\). The post-conditions of the composition method \(m_i \land m_j\) are the post-conditions of the individual methods \(m_i\) and \(m_j\).

The \textit{sequence} operator, represented as \(m_i; m_j\), denotes the execution of the two methods \(m_i\) and \(m_j\) in sequence, wherein \texttt{env} must satisfy the pre-condition of \(m_i\), and the post-condition of \(m_i\) 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 \textit{choice} operator, represented as \( m_i \lor m_j \), denotes that the composition method consists of the semantics of either the method \( m_i \) or the method \( m_j \) (not both). The pre- and post-conditions for this composition method are the pre- and post-conditions of the chosen method \( m_i \) or \( m_j \) (not both).

The \textit{pipe} operator, represented as \( m_i \; | \; m_j \), denotes the execution of the two methods \( m_i \) and \( m_j \) in sequence wherein the output of the method \( m_i \) is the input of the method \( m_j \). The \textit{env} must satisfy the pre-condition of \( m_i \), and the post-condition of \( m_i \) 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 \textit{loop} operator, represented as \( m_i^{\ast} \), 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 Complex Composition Operators

Primitive manipulation and composition operators as defined in Section 2 provide the basic manipulation and composition at the level of attributes and methods. Complex operators can, in general, be formulated via a sequence of primitive operators. However, we find that to increase the ease of use of the overall system and to provide better system efficiency it is often desirable to extend this basic set of primitive operators to now include a set of more complex operators that provide manipulation and composition at the level of classes and components. We therefore provide a set of commonly used \textit{complex manipulation and composition operators} as part of the system.

3.1 Complex Manipulation Operators

The public interface of a component typically comprises of \textit{properties}, \textit{events} and \textit{methods} [9, 12]. Properties and events are recognized only if appropriate methods that conform to the specification of a component model are defined. To enable the manipulation of the properties and events, we now provide a set of complex manipulation operators. This set includes the operators \textit{configureProperty}, \textit{addListener}, \textit{addProperty}, \textit{linkProperty}, \textit{addEvent}, and \textit{linkEvent}. The \textit{configureProperty} operator configures the property of a component. The \textit{addListener} operator registers the \textit{event listener} component to an \textit{event source} component based on a specified event. The \textit{addProperty} operator adds a new property \( p \) to the component. The \textit{linkProperty} operator creates a new property \( p \) in a component, such that the property \( p \) represents the property \( p \) of an underlying component. The \textit{addEvent} operator adds a new event \( e \) for the specified component. Lastly, the \textit{linkEvent} create a new event \( e \) in a component such that the new event \( e \) represents the event \( e \) of an underlying components.

Each of these complex manipulation operators can be expressed by a sequence of primitive operators. As an example, Figure 1 illustrates the \textit{addListener} operator specified using the primitive manipulation operator \textit{invokeMethod}. Here the \textit{invokeMethod} operator is used to invoke the method with label

5
def addListener(eventSource, listenType, eventListener) =
    pre: interface($eventListener) == $listenType
    $metName = "add" + $listenType
    invokeMethod($eventSource, $metName, null, $eventListener)
    post: -

Figure 1: The addListener Operator.

listenType\(^2\) in the component eventSource. The eventListener, a parameter passed into the method listenType, represents the component instance listening to the eventSource for an event specified in listenType.

3.2 Complex Composition Operators

Composition at higher granularities, that is at the component level can be classified into two categories, connection-oriented composition and aggregation-based composition. Connection-oriented composition \([2]\) specifies how components are plugged together via the mechanisms such as events or pipes and filters. Aggregation-based composition \([2]\), on the other hand, specifies the aggregation of components to a higher-level component. To enable such composition at the level of components we define two new complex composition operators, addConnection and addToContainer. The addConnection operator registers a set of components, event listeners, to a specified component, event source, creating a connection between the components such that when the event source fires a specified event, the defined actions in the event listeners are invoked. The addToContainer operator, on the other hand, targets aggregation-based composition, and creates a hierarchy between two specified components, the parent and the child component. Here the child component is aggregated in the parent component.

Each of these operators, provided as a convenience to the user, can in fact be accomplished via a sequence of primitive composition operators. As an example, Figure 2 shows how the addConnection operator can be specified using the primitive operators. Here, the interface of the eventListener is first checked to see whether it matches the listenType. If a match is found, the addListener operator is invoked to form a connection between the eventSource and the event listeners. If they are

---

\(^2\)For a specific component model, such as JavaBean, some string manipulation may be required to produce the correct label.
mismatched, then a new class must be created along with the appropriate method and “listener” connection.

4 Composition Patterns and Templates

Composition operators, primitive and complex, as defined in Sections 2 and 3 while necessary, represent a finite set of operations. This set of finite operators is however, not sufficient to address the rich diversity and possibly infinite number of feasible compositions at the class and composition levels. To account for this growing need for flexibility in composition semantics, we now introduce the notion of (1) composition patterns, a flexible mechanism for the specification of high level component compositions; and (2) composition pattern templates, an extension of composition patterns that now provide re-usability of component compositions.

4.1 Composition Pattern

A composition pattern, much like the complex composition operators in Section 3.2, is defined via a sequence of composition operators, both primitive and complex. Consider for example the pictorial representation of the composition of four components in Figure 3. Here the components, two lists, sourceList, pickedList, and two buttons, select and deSelect, are composed together to form the application SelectDeSelect. On click of the select button, the selected item from the sourceList is moved to pickedList. Similarly, on deSelect the selected item from the pickedList is moved back to sourceList. To accomplish this composition of components, the events onClick in both the select and deSelect buttons must be “connected” to the appropriate methods in the components sourceList and pickedList.

Figure 4 depicts a composition pattern that accomplishes the composition of the four components as illustrated in Figure 3. The first step of this composition is to instantiate five components, sourceList, pickedList, select and deSelect, and the frame, a container for the other four components. This is done via the addComponent operator (Section 2). The second step is to establish the aggregation-based composition of the first four components into the frame component. This is done via the addToContainer composition operator (Section 3). The configureProperty operator is then used to configure the properties of the select and deSelect buttons. The last step of this composition is to establish the connection between the select button and the two lists, the sourceList and pickedList, as well as the connection between the deSelect button and the two lists, the sourceList and pickedList. This is accomplished via the addConnection operator. Here the select and deSelect buttons are the event sources while sourceList and pickedList are the event listeners.

4.2 Composition Pattern Templates

Composition patterns such as the one shown in Figure 4 are complex composition operations that are written once and used once. These patterns are bound to the classes and components 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
Figure 3: A Select and DeSelect Example

```
SelectDeSelect {
    addComponent ("java.awt.Frame", "frame")
    addComponent ("java.awt.List", "sourceList")
    addComponent ("java.awt.List", "pickedList")
    addComponent ("java.awt.Button", "select")
    addComponent ("java.awt.Button", "deSelect")
    addToContainer ("frame", "sourceList")
    addToContainer ("frame", "pickedList")
    addToContainer ("frame", "select")
    addToContainer ("frame", "deSelect")
    invokeMethod ("sourceList", "fillKidList", "sample.txt")
    configureProperty ("select", "label", "Select")
    configureProperty ("deSelect", "label", "DeSelect")
    $post = (pickedList.size() + sourceList.size()) ==
             (pickedList'.size() + sourceList'.size())
    $actSelect = (sourceList.getSelectedValue() |
                   pickedList.addElement())
                   sourceList.removeElement()
    addConnection ("select", "mouseListener", "onClick",
                   "{sourceList, pickedList}", $actSelect,
                   true, $post)
    $actDeselect = (pickedList.getSelectedValue() |
                    sourceList.addElement());
    pickedList.removeElement();
    addConnection ("deSelect", "mouseListener", "onClick",
                   "{pickedList, sourceList}", $actDeselect,
                   true, $post)
}
```

Figure 4: A Select and DeSelect Composition Pattern.

inline of one component into another, merge of two components, concat of two components, or diff of two components, 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 primitive and complex 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 the 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 input parameters.

Composition pattern templates are, thus, named, parameterized, and generalized composition patterns. Figure 5 illustrates the templated version of the composition pattern in Figure 4. Here the template is assigned a name SelectDeSelectTemplate and is parameterized. The SelectDeSelectTemplate takes three input parameters, fileName, selectLabel and deSelectLabel, where the fileName represents the information to be added into the sourceList, the selectLabel and the deSelectLabel provide the labels for the buttons select and deSelect and are used to configure the properties of the
SelectDeSelectTemplate(fileName, selectLabel, deSelectLabel) {
    addComponent ("java.awt.Frame", "frame")
    addComponent ("java.awt.List", "sourceList")
    addComponent ("java.awt.List", "pickedList")
    addComponent ("java.awt.Button", "select")
    addComponent ("java.awt.Button", "deSelect")
    addToContainer ("frame", "sourceList")
    addToContainer ("frame", "pickedList")
    addToContainer ("frame", "select")
    addToContainer ("frame", "deSelect")
    invokeMethod ("sourceList", "fillKidList", $fileName)
    configureProperty ("select", "label", $selectLabel)
    configureProperty ("deSelect", "label", $deSelectLabel)
    $post = (pickedList.size() + sourceList.size()) ==
    (pickedList'.size() + sourceList'.size())
    $actSelect = (sourceList.getSelectedValue() |
                  pickedList.addElement()) |
                sourceList.removeElement()
    addConnection ("select", "mouseListener", "onClick",
                   "{sourceList, pickedList}", $actSelect,
                   true, $post)
    $actDeselect = (pickedList.getSelectedValue() |
                   source.addElement()) |
                  pickedList.removeElement()"
    addConnection ("deSelect", "mouseListener", "onClick",
                   "{pickedList, sourceList}", $actDeselect,
                   true, $post)
}

Figure 5: A Select and Deselect Component Pattern Template

two buttons respectively. This template can now be instantiated with the parameters “sample.txt”,
“select” and “deSelect” to produce the composition pattern shown in Figure 4.

However, not all templates can be accomplished by a simple sequence of primitive or complex compo-
sition operators. Some template compositions may require looping over all the properties of a given com-
ponent, or iteration over its events. To clarify this, consider the composition pattern in Figure 6 that depicts
the merge of the components list and selector to form a composite component ListMultiselector. The
composite component, ListMultiselector, contains all events defined in the list and selector components. Additionally, a new event close is defined as a sequence of close() events of the select
and list components. Figure 7 presents the templated version of the composition pattern in Figure 6.
Here, in contrast to the template SelectDeSelectTemplate in Figure 5, we use iterative constructs,
for, to loop over all events of the master and client components, and to subsequently link the events
of the master and the client components.

Instantiation of a Composition Template Composition templates provide a rich resource for the users
by encapsulating commonly used composition patterns. A user may thus select a composition template,
 instantiate it and produce a composition pattern which can then be executed. The goal of the instantiation
**ListMultiselector**

```java
ListMultiselector {
    addComponent ("java.awt.List", "list")
    addComponent ("java.awt.Multiselector", "selector")
   addToContainer ("list", "selector")
    linkEvent ("paint", "list.paint")
    linkEvent ("select", "selector.select")
    linkEvent ("deselect", "selector.deselect")
    linkEvent ("close", "selector.close; list.close")
}
```

Figure 6: List and Multiselector Composition Pattern

**MergeMasterClient**

```java
MergeMasterClient (master, client){
    addComponent ("$master", "container")
    addComponent ("$client", "part")
    addToContainer ("container", "part")
    let $tmp = event("container") \ event("part")
    for all $e ∈ event("container") and $e ∉ $tmp
        linkEvent ($e, "container".$e)
    for all $e ∈ event("part") and $e ∉ $tmp
        linkEvent ($e, "part".$e)
    for all $e ∈ $tmp
        linkEvent ($e, {"part".$e;"container".$e})
}
```

Figure 7: List and Multiselector Composition Pattern Template

is thus to produce a flat composition pattern that does not contain any conditional or iteration statements, but is rather a simple sequence of the composition operators. Thus, the instantiation phase of a composition template must provide (1) binding of the parameters to the variables; (2) removal of the conditional and iteration statements. For example, all for loops in the Figure reftime-template-list-multi must be replaced by a sequence of the actual linkEvent operators. This replacement of the for loops is done based on the information collected by querying the component schema; and (3) the replacement of any embedded composition template by an equivalent composition pattern. The pure composition pattern can then be executed to produce a component schema for the composed component.

### 5 Composition Correctness

Bertrand Meyer states that “Correctness is the measure of prime quality. If the system does not do what it is supposed to do everything else about it - whether it is fast, has a nice user interface - matters little” [14]. Meyer has advocated the use of contracts [14] as a mechanism to specify a contract, a set of pre and post-conditions that must be met to ensure the correctness of a method. To ensure the correctness of composition patterns while dealing with its inherent complexity, we now provide a hierarchy of contracts that must be satisfied to ensure both the correctness of the composition and the correctness of the resultant application or component. In this section, we focus on defining the different levels of contracts and provide some intuition on how they can be used to verify correctness.

#### 5.1 Contracts

Specification can be represented as contract [14] between the supplier of a certain service and the client of that service. As in [14], we use pre-condition and post-condition to describe the contract. The pre-condition defines the condition under which a call to a method is legitimate, and the post-condition defines conditions that must be ensured by a method on return. Pre- and post-conditions are described based on first order predicate logic (FOPL).
We provide two levels of contracts, at the method level as well as at the level of the entire composition. We term these method contract and composition contract respectively. As an example consider the following contract: \( \{\text{pre}_A\} \ A \ \{\text{post}_A\} \). Here \( \text{pre}_A \) and \( \text{post}_A \) denote the pre- and post-conditions of the method \( A \). On the other hand consider the composition contract for the composition of two methods \( A \) and \( B \), using any composition operator, \( \{\text{pre}_\text{compose}\} \ \{\text{pre}_A\} \ A \ \{\text{post}_A\} \ \text{op}\ \{\text{pre}_B\} \ B \ \{\text{post}_B\} \ \{\text{post}_\text{compose}\} \).

A composition is said to be correct, composition correctness, if both the method contracts for all methods in the composition and the composition contract are satisfied. The composition is said to be incorrect otherwise.

### 5.2 Verification of Composition

Our main intuition for verification of composition patterns is based on the hypothesis that if all parts of the composition are correct, then the composition itself must be correct. That is to say that much as composition patterns and templates are composed using primitive composition operators, verification of composition patterns can also be accomplished via the sequential verification of all primitive composition operators.

Consider the following contract.

\[
S \ \{\text{pre}_\text{compose}\} \ \{\text{pre}_A\} \ A \ \{\text{post}_A\} \ S' \ \text{op}\ \{\text{pre}_B\} \ B \ \{\text{post}_B\} \ S'' \ \{\text{post}_\text{compose}\}
\]

This is similar to the above composition contract between the methods \( A \) and \( B \) with the exception that there are three additional states: \( S \)-before, \( S' \)-in-between and \( S'' \)-after. \( S \) represents the state before the composition; \( S' \) represents the state after the first method invocation is completed and before the second starts; and \( S'' \) represents the state after the last method invocation is completed. While the verification process itself is similar, the states of the systems itself vary with the semantics of the each primitive composition operator (conjunction, sequence, choice, pipe and loop).

Figure 8-12 depict the combination of the verifications that result due to the semantics of the primitive operators. As an example, consider Figure 8. Recall that a conjunction operator, represented as \( m_i \land m_j \), denotes the execution of the two methods \( m_i \) and \( m_j \) simultaneously wherein the initial state of the system, \( \text{env} \), must satisfy the pre-conditions of both methods \( m_i \) and \( m_j \). The post-conditions of the composition method \( m_i \land m_j \) are the post-conditions of the individual methods \( m_i \) and \( m_j \). Thus, the composition of the two methods \( A \) and \( B \) via the conjunction operator is valid only if (1) the state \( S \) satisfies the three pre-conditions: \( \text{pre}_\text{compose} \), \( \text{pre}_A \) and \( \text{pre}_B \); and (2) the state \( S'' \) satisfies the post-condition \( \text{post}_\text{compose} \) where \( S'' \) results from the update of the state \( S \) with two post-conditions, \( \text{post}_A \) and \( \text{post}_B \).

```
1. \( S \rightarrow \text{pre}_\text{compose} \) and \( \text{pre}_A \) and \( \text{pre}_B \)
2. \( S'' \rightarrow \text{post}_\text{compose} \)
Note: \( S'' \equiv \text{Update}(S, \text{post}_A, \text{post}_B) \)
```

Figure 8: Composition Correctness for Conjunction

```
1. \( S \rightarrow \text{pre}_\text{compose} \) and \( \text{pre}_A \)
2. \( S' \rightarrow \text{pre}_B \)
3. \( S'' \rightarrow \text{post}_\text{compose} \)
Note: \( S'' \equiv \text{Update}(S', \text{post}_A) \) and \( S'' \equiv \text{Update}(S', \text{post}_B) \)
```

Figure 9: Composition Correctness for Sequence
1. $S \rightarrow (pre_{compose} \text{ and } pre_A)$ or $(pre_{compose} \text{ and } pre_B)$
2. $S'' \rightarrow post_{compose}$
Note: $S'' \equiv Update(S, post_A, post_B)$

Figure 10: Composition Correctness for Choice

1. $S \rightarrow pre_{compose}$ and $pre_A$
2. $S' \rightarrow pre_B$
3. $S'' \rightarrow post_{compose}$
Note: $S' \equiv Update(S, post_A)$ and $S'' \equiv Update(S', post_B)$

Figure 11: Composition Correctness for Pipe

6 XCompose: A Component Composition Framework

To validate the ideas presented in this paper, we have developed XCompose, an XML-based component composition framework, that operates as a thin-composition layer on top of existing component models. The XCompose system (1) takes as input the component schemas of the components participating in the composition; (2) provides for user interaction by allowing the user to either (a) compose a new composition pattern; (b) instantiate an existing composition pattern; or (c) edit and subsequently instantiate an existing composition pattern; and (3) produces as output a component schema for the newly composed component or application. In this section we first provide a discussion on the two main choices that we have made in our framework, and then provide an overview of the XCompose architecture.

6.1 Our Choices: Component Schemas and Composition Language

The two main choices that we had to make in the design of the XCompose framework were that of the language and format of the component schemas, and the composition language used for the composition templates. We now provide a brief discussion of the choices that we made.

6.1.1 XDescriptors - The Component Schemas

While the composition patterns and templates as described in Section 4 provide for the flexibility and the reusability of the component compositions, they do not provide portability for these compositions. To achieve this platform as well as component model independence, we require that a component must expose its “component schema”. Each component has an inherent component schema that describes (1) its public interface such as its set of properties, set of events and set of methods; and (2) its architectural layout, that is it describes how the underlying classes in the component are connected. Furthermore, we require that the component schema of the underlying component be expressed in a uniform manner. Given the wide acceptance of the XML model [8] and its suitability to express the semi-structured nature of a component composition, we use XML as the unifying model for the XCompose system. Thus, in the XCompose framework the component schemas are expressed as XML documents that conform to a pre-set XML Schema. These XML
6.1.2 XQuery - The Composition Language

Based on the template examples (Figure 5 and 7), we now first examine the requirements of a composition language that provides the glue logic to enable the specification of composition templates. The language should allow users to combine the composition operators to formulate arbitrarily complex composition templates based on some underlying components. The most rudimentary requirements for such a language are: iteration, the ability to iterate over the public interface data - the properties, events and methods of documents (component schemas) are termed XDescriptors.

Figure 13 gives the general format of an XDescriptor document. The XDescriptor consists of five main elements, class, properties, events, methods and composition, where the element class represents the component itself; the element properties represents a set of properties wherein both property name and type of each property are given; the element events represents a set of events wherein name and the required interface of each event are specified; the element methods represents a set of methods wherein method name, the return type and a list of parameter types of each method are provided; and the element composition represents the composition of the component. Figure 14 shows a fragment of the XDescriptor produced for the SelectDeSelect composition pattern shown in Figure 4.

Figure 13: XDescriptor - An XML-based Component Schema.

Figure 14: XDescriptor - A Select and Deselect Application
the component; *conditional statements*, to allow for conditional statements (*if then else* construct); *existential and universal quantification*, to allow checks for some or all entities of a set; and *type system*, it must be strongly typed. In addition, the language must allow variable binding, path manipulation, and must be portable - platform independent. These language requirements match the requirements of a composition language defined in [16, 2] and meet the requirements for composition code reuse and extensibility.

To translate a composition pattern template to a composition pattern, we must necessarily have access to the component schemas, the XDescriptors, that describe the public interface of the component and its architectural layout. Thus, to further support re-usable composition pattern templates, a composition language must additionally provide constructs to access and query the XDescriptors.

We have found that XQuery [10] fulfills all of these requirements for a composition language. Hence our language of choice for the specification of composition templates is XQuery [10].

6.2 An Architectural Overview

Figure 15 gives the overall architecture of the XCompose framework. The top third of the figure depicts the core of the XCompose framework.

Figure 15: Architecture of XCompose - An XML-Based Component Composition Framework.

The core of the XCompose framework is the Composition Manager that provides a fixed set of composition operators (refer Section 2) as well as the ability to create and verify composition patterns (refer Section 4.1) via three main modules: composition editor, instantiation tool and pattern verifier. The
Composition Editor takes an input one or more XDescriptor, and allows the users to browse existing XDescriptors, composition templates, or design their own composition pattern or composition templates using the set of composition operators provided by the system. The user may also instantiate or verify an existing composition template by providing the requisite input parameters and passing it through the Instantiation Tool or the Pattern Verifier respectively. The Instantiation Tool uses an XQuery engine [10] to execute the embedded XQuery statements in the composition template against the XDescriptors to produce a composition pattern as output. The Pattern Verifier takes as input a composition pattern and verifies that the composition is indeed a valid composition and conforms to the contracts that have been specified. The composition pattern can then be executed by the Execution Engine that outputs an XDescriptor for the composed component. A Deployment Engine can then be used to translate the XDescriptor into a set of component model specific classes and thus in fact deploy the composed component.

Composition patterns of both composite components and applications 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. The middle third of the figure (Figure 15) depicts the translation layer. To facilitate composition of components independent of the component model, we require that the component schemas of each specific component model be translated into XDescriptors. We provide an XDescriptor Generator tool that can generate the XDescriptors for a given component. Currently, we only support JavaBeans [15] but plan on extending this to other models in the future. The bottom third of the architecture figure (Figure 15) depicts the underlying component models that are independent from the XCompose framework.

7 Related Work

There are many parallels that can be drawn between XCompose and our previous work, SERF [6, 7]. In this 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. 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 [5], based on the ODMG [5] 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.

XDescriptor is influenced by Bean Markup Language, BML [20], and Component Markup Language, CoML [2]. Both BML and CoML present a composition language based on XML syntax. BML focuses on JavaBean Composition, while CoML aims at the composition independent of the component model. Clearly, while there are some similarities between their work and ours, XDescriptor in our case does not serve as a new composition language but rather as enabling technology to facilitate the component composition.
8 Conclusion

In this paper, we propose an XML-based component composition framework, XCompose, to provide flexibility, extensibility, re-usability and correctness of component composition. The XCompose framework is based on the hypothesis that a set of operators can be combined to produce arbitrarily complex components or applications. We term these composition patterns. To provide the notion of re-usability of the composition patterns, we introduce the notion of composition pattern templates, wherein the composition patterns can be generalized via the use of a query language and saved in a template library for later re-use. A key advantage of our work is the provable correctness of the composition patterns via composition contracts.

To summarize, our work makes the following contributions:

- Basic composition operators with minimal semantics to ensure correct composition of composite components or applications.
- Composition pattern to enable flexibility and extensibility of the component composition.
- Composition pattern template to enable re-usability of the compositions.
- Composition contracts to enable verification of the component compositions.
- XCompose framework that implements our core ideas and provides flexibility, extensibility, re-usability and correctness of component compositions.

Currently, the XCompose framework is under development as a thin-layer of composition functionality on top of existing component models. We use XML as our middle-layer model to express the descriptions of the components and make use of the XQuery engine to access such descriptions.

References


