COCO: COMPOSITION MODEL AND COMPOSITION MODEL IMPLEMENTATION

Naiyana Tansalarak and Kajal T. Claypool
University of Massachusetts - Lowell
1 University Ave. Lowell, MA 01854

Keywords: Component model, Component Model Implementation, Composition Model, Composition Model Implementation, Composition Style, Composition Script, Composition Operator, Unifying Component Model.

Abstract: The success of component-based development has been impeded by interoperability concerns, including *component model*, *semantic*, *syntactic*, *design* and *platform* incompatibilities, that often come into play when composing two or more independently developed components. In this paper we propose a *CoCo composition model* that elevates compositions to first class citizenship status. The model defines a standard for describing the composition of components transparent to any underlying incompatibilities between the collaborating components. We also present a *CoCo composition model implementation* that provides the required support to describe and subsequently execute the composition to produce an executable application. We advocate the use of XML Schemas as a mechanism to support this composition model. To support the composition model implementation we provide (1) a taxonomy of primitive composition operators to describe the *connection* between components; (2) XML documents as a description *language* for the compositions; and (3) the development of a set of deployment plugins that address any incompatibilities and enable the generation of the model-specific and platform-specific applications.

1 INTRODUCTION

Component-based software engineering attempts to address the ever increasing demand for new software applications by enabling a compositional approach to software construction in which applications are built from pre-fabricated components, rather than developed from scratch (Heineman and Councill, 2001). A number of component models (with corresponding implementations) have been defined to date and many have been widely adopted in practice. Examples of component models are CORBA (Siegel, 1996), JavaBeans (Muller and Davidson, 1996), Enterprise JavaBeans (Roth, 1998), COM (Box, 1998), and .NET (Chappell, 2002). These different component models have stimulated the rapid development of components by different developers, with the hope that eventually most components needed for application building will be available as off-the-shelf components.

However, the success of component-based development has been impeded by interoperability concerns that often come into play when composing two or more independently developed components (Gschwind et al., 2002; Vallecillo et al., 2000; Yakimovich et al.,). These concerns encompass *component model* incompatibilities that occur when the *to-be composed* components are developed based on the requirements of disparate component models; *syntactic* incompatibilities that arise when there are signature or interface mismatches between the *to-be composed* components; *semantic* incompatibilities that typically occur when the behavior expected by one component (the client component) as specified by the "design by contract" principle is incompatible with the behavior provided by the other (server) component; *design* incompatibilities that occur when there is an architectural or a design level mismatch between the *to-be composed* components; and lastly *platform* incompatibilities that come into play when a component is constructed on one platform but the execution infrastructure supports a different platform. Interoperability and hence composability of two or more components may be restricted by one or more of these incompatibilities, requiring in some cases glue code to enable the collaborative operation of two components, while in other cases completely occluding the inter-operation of the given components.

Tansalarak N. and T. Claypool K. (2005).
COCO: COMPOSITION MODEL AND COMPOSITION MODEL IMPLEMENTATION.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 340-345
DOI: 10.5220/0002525503400345

Prior research (Raje et al., 2001; Oberleitner et al., 2003; America, 1990; Garlan et al., 2000; Shaw et al., 1995; Allen and Garlan, 1997) focusing on interoperability of components has investigated possible solutions to these incompatibilities singularly, rather than as a complete set of concerns that must be addressed to provide comprehensive support for the composition of components. We propose a *CoCo composition model* that elevates compositions to first class citizenship status and defines the standard for describing the composition of components transparent to any underlying incompatibilities between the collaborating components. Figure 1 shows a high level view of the composition model that classifies information into three primary categories: (i) an *object model* - that defines the selection, instantiation, configuration and initialization of a set of underlying components in the system; (ii) an *interface model* - that defines the interface of a resultant composite component (or application) to enable the resultant composite component (or application) itself to interact with the other components (or applications); and (iii) an *association model* - that defines how components are composed via both connection-oriented and aggregation-based compositions. Details on the composition model can be found in (Tansalarak and Claypool, 2004).

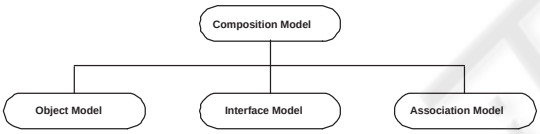

Figure 1: The Composition Model

In this paper, we focus on the *CoCo composition model implementation* that provides the required support to describe and subsequently execute the composition to produce a composed application. To support the CoCo composition model implementation we provide (1) a taxonomy of primitive composition operators to describe the *connection* between components; (2) XML documents as a description *language* for the compositions; and (3) a set of deployment plugins that address any incompatibilities and enable the generation of the composed application (or composite component) in different languages and component models as well as on different platforms. We exploit the inherent tight coupling between an XML Schema and an XML document to provide the same coupling between the composition model (expressed as an XML Schema) and the composition model implementation enabling us to validate and check the conformance of a composition against the composition model (or a core subset of it).

Roadmap: The rest of paper is organized as follows. Based on the GUI composition style, Section 2 presents the set of primitive composition operators, while Section 3 illustrates the description of compositions using an example. To complete the CoCo composition model implementation we show in Section 4 the generation of deployable applications based on the described compositions. We conclude in Section 5.

2 PRIMITIVE COMPOSITION OPERATORS

In literature (Garlan et al., 2000; Allen and Garlan, 1997; Shaw et al., 1995; Achermann, 2002; Achermann et al., 2001) *connectors* have played an essential role in mediating interactions between the underlying components of the system. Additionally we believe that to provide comprehensive support for component compositions, an *order* concept that provides a sequencing of component interactions is essential. Based on these, we now introduce a set of *composition operators* as the core construct of CoCo composition model implementation. These *composition operators* represent the building blocks that can be used in the instantiations of the composition model, and on the basis of which arbitrarily complex component compositions can be defined. In this section, we describe the set of primitive *method* composition operators as well as the set of primitive *event* and *container* composition operators that are at the heart of the CoCo composition model implementation.

2.1 Method Composition Operators

Butler et al. (Butler and Duke, 1998) have defined a set of operators for combining object interactions at the granularity of a method. Given that most component compositions are also accomplished at the method level, we now define a set of *method composition operators* to enable composition of methods from one or more components. This set of method composition operators consists of:

- 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.
- 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.
- The *choice* operator, represented as $\mathtt{m}_i \vee \mathtt{m}_j$, denotes the execution of either the method $\mathtt{m}_i$ or the method $\mathtt{m}_j$, but 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 *loop* event operator, represented as $\mathtt{m}_i^*$, denotes the repeated consecutive execution of the method $\mathtt{m}_i$.

2.2 Event Composition Operators

In GUI composition style, the firing of an event from an event component may trigger interaction(s) from one or more listener component(s) at the granularity of method. We thus define an *event composition operator*, denoted as $\mathtt{e} \rightsquigarrow \mathtt{m}$, to represent the relationship between an event $\mathtt{e}$ and a set of method interaction(s) $\mathtt{m}$. $\mathtt{m}$ thus represents either a simply method invocation of a listener component, $\mathtt{m}_i$, or the composition of method invocations from a set of listener components as defined in Section 2.1, $\{ \mathtt{m}_i \wedge \mathtt{m}_j, \mathtt{m}_i ; \mathtt{m}_j, \mathtt{m}_i \vee \mathtt{m}_j, \mathtt{m}_i \mid \mathtt{m}_j$, and $\mathtt{m}_i^* \}$.

2.3 Container Composition Operators

We now define a set of *container* composition operators to facilitate the aggregation and display of the underlying components. This set of container composition operators consist of:

- The *position* container operator, represented as $\mathtt{c}_i \ll \mathtt{c}_j$, specifies that the component instance $\mathtt{c}_j$ be positioned within the component instance $\mathtt{c}_i$ at a default location chosen by the layout manager of the component instance $\mathtt{c}_i$. For example, the component `BorderLayout` provides the default locations to position its underlying components from left to right and then from top to bottom.
- The *position-at* container operator, represented as $\mathtt{c}_i \ll \mathtt{c}_j$ @ `location`, specifies that the component instance $\mathtt{c}_j$ be positioned within the component instance $\mathtt{c}_i$ at the specified `location`. The `location` must conform to the layout management defined in the component instance $\mathtt{c}_i$. For example, the component `BorderLayout` provides five locations to position its underlying components: *North*, *South*, *West*, *East*, and *Center*.

3 A COMPOSITION SCRIPT EXAMPLE

The eventual goal of a CoCo composition model implementation is to enable component-based construction of a composite component or an application based on a specified composition model. While the composition operators presented in Section 2 provide the semantics for combining individual methods, they cannot singularly express complete compositions. We now introduce composition scripts, that provide the necessary glue logic to tie these operators together and to comprehensively describe a composition, using XML documents.

In this section, we illustrate how an XML-based composition script can be used to describe the composition of an application. Figure 2 pictorially represents the *SpeedLogo* application. The *SpeedLogo* application is composed from two components `Logo` and `SliderFieldPanel`. The component `Logo` is a primitive component while the component `SliderFieldPanel` is mainly constructed from two primitive components `JSlider` and `JTextField`. When the `JSlider` value changes, the `JTextField` value is automatically updated with the new value. On the other hand, when a new value is entered in the `JTextField` and the user presses the *Enter* key, the `JSlider` is automatically repositioned to the appropriate location. Overall in the *SpeedLogo* application, when the `SliderFieldPanel` value changes, the animation speed of `Logo` is changed. The two components `Logo` and `SliderFieldPanel` are aggregated into the component `JFrame` at `Center` and `South` locations, respectively.

Figure 2: The *SpeedLogo* Application

The partial composition script depicting the semantics and the layout for the construction of the *SpeedLogo* is shown in Figure 3. This script, that conforms to the GUI composition model (Tansalarak and Claypool, 2004), incorporates the composition operators defined in Section 2. For example, on line `072` the <`op type` = "|"> is used to ensure that the output returned from the method `getCurrentValue` of the component instance `sliderField` is the input of the method `setAnimationRate` of the component instance `logo`.

The overall architecture of the *SpeedLogo* application (Figure 4) can be efficiently and effectively extracted from the composition script shown in Figure 3. Here, a *box* denotes a component instance as well as the component library that instantiates the component instance, an *arrowed line* the connection-oriented

composition, a *double-arrowed line* the aggregation-based composition, a *dash line* the hierarchical structure, a *rectangle* the property, and *oval* the event.

```
000. <?xml version="1.0"?>
001. <GUIscript>
002. <application>
003.     <name >SpeedLogo</name>
004.     <compInstances>
         .....
033.       <compInstance role="Client">
034.         <comp>javax.composite.SliderFieldPanel
             </comp>
035.         <cid>sliderField</cid>
036.         <configuration>
037.           <configProperty>
038.             <pName>currentValue</pName>
039.             <pValue>
040.               <const>10</const>
041.             </pValue>
042.           </configProperty>
         .....
053.     <compositions>
054.       <eCompositions>
055.         <eComposition>
               .....
061.           <eCompInstances>
062.             <eCompInstance>
063.               <rid>sliderField</rid>
064.               <event>PropertyChange</event>
065.               <eAction>propertyChange</eAction>
066.             </eCompInstance>
067.           </eCompInstances>
068.           <lCompositions>
069.             <lCompInstance>
070.               <rid>sliderField</rid>
071.               <callMethod>getCurrentValue
                   </callMethod>
072.               <op type="|" />
073.             </lCompInstance>
074.             <lCompInstance>
075.               <rid>logo</rid>
076.               <callMethod>setAnimationRate
                   </callMethod>
077.             </lCompInstance>
078.           </lCompositions>
       .....
099. </application>
100. </GUIscript>
```

Figure 3: The Partial `SpeedLogo` Composition Script

4 DEPLOYMENT

The last step of the composition step process is the conversion of the script to a deployable application or component that conforms to the desired platform, model, and language. We present a set of manipulation operators that provide an intermediate unifying

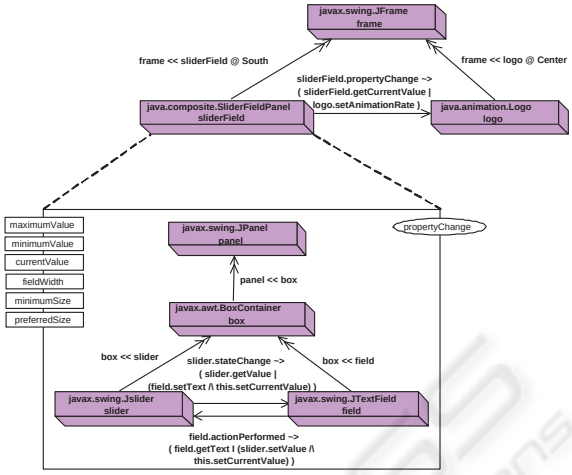

Figure 4: The Overall Architecture of the *SpeedLogo* Application

format between the XML-based composition scripts and the final application written in the programming language of choice. These unifying manipulation operators together with specific code generation plugins provide the instrument for managing language, component model and platform incompatibilities. Additional glue logic can be provided to support handling of syntactic incompatibilities as part of the deployment process. We do not currently address semantic and design incompatibilities at deployment time.

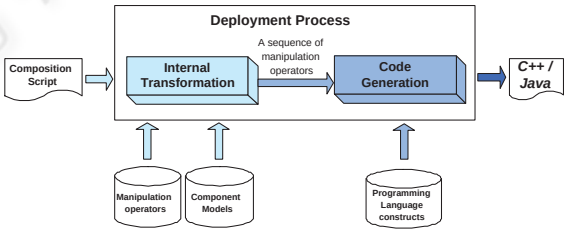

Figure 5: The Deployment Process

Figure 5 pictorially depicts the deployment process, consisting of two essential modules: *internal transformation* and *language transformation* modules. The internal transformation module provides the mapping of the composition script to a sequence of corresponding manipulation operators. We define six primitive manipulation operators (Tansalarak and Claypool, 2004), namely *AA* - to add an attribute, *AM* - to add a method, *IM* - to invoke a method, *IN* - to instantiate an attribute, *ET* - to extend a composition, and *CC* - to create an inner class. Figure 6 gives the manipulation operator equivalent of the *SpeedLogo* composition script given in Figure 3. Consider as an

example the composition fragment from lines `033` to `044`. This fragment specifies the instantiation of the `SliderFieldPanel` and the configuration of its property `currentValue`. This is translated into the following operators: `IN ("sliderField", "java.composite.SliderFieldPanel")` and `IM ("sliderField", "setCurrentValue", "void", <"10">)`.

```
ET ("javax.swing.JFrame");
IM ("this", "setTitle", "void", <"SpeedLogo Animation">);
AA ("container", "private", "java.awt.Container", "1", "null", "local");
IM ("this", "getContentPane", "container", <>);
AA ("border", "private", "java.awt.BorderLayout", "1", "null", "local");
IN ("border", "java.awt.BorderLayout");
IM ("container", "setLayout", "void", <"border">);
AA ("color", "private", "java.awt.Color", "1", "null", "local");
IN ("color". "java.awt.Color", <"0xeeeeee">);
IM ("this", "setBackground", "void", <"color">);
AA ("sliderField", "private", "java.composite.SliderFieldPanel", "1",
    "null", "global");
IN ("sliderField", "java.composite.SliderFieldPanel");
IM ("sliderField", "setCurrentValue", "void", <"10">);
AA ("logo", "private", "java.animation.Logo", "1", "null", "global");
IN ("logo", "java.animation.Logo");
IM ("logo", "startAnimation", "void", <>);
IM ("sliderField", "addPropertyChangeListener", "void",
        <"new sliderField_PropertyChange()">);
CC ("sliderField_PropertyChange", "java.beans.PropertyChangeListener",
    <>,
    < AM ("propertyChange", "public", "void",
        <"java.beans.PropertyChangeEvent e">, "null", "null",
        { AA ("value", "private", "int", "1", "null", "local");
          IM ("sliderFieldPanel", "getCurrentValue", "value", <>);
          IM ("logo", "setAnimationRate", "void", <"value">) ; }
    >);
IM ("container", "add", "void", <"logo", "BorderLayout.CENTER">);
IM ("container", "add", "void", <"sliderField", "BorderLayout.SOUTH">);
```

Figure 6: The Sequence of Manipulation Operators for the *SpeedLogo* Composition Script

Furthermore, the syntactic incompatibilities between components are handled within the internal transformation module. Consider, for example, the connection-based composition between the component instances `sliderField` and `logo` described in lines `056` to `080` of Figure 3 wherein the syntactic incompatibility [1] between these two component is transparent. This handler can be accomplished by creaing an inner class that implements the required interface (Stearns, 2001; Akerley et al., 1999).

As a next step, the manipulation operators are translated into the desired programming language via code generation plugins. Figure 7 gives the Java equivalent of the sequence of manipulation operators given in Figure 6 and the composition script given in Figure 3.

```
public class SpeedLogo extends javax.swing.JFrame {
        private java.animation.Logo logo;
        private java.composite.SliderFieldPanel sliderField;

        public SpeedLogo () {
            java.awt.BorderLayout border;
            java.awt.Container container;
            java.awt.Color color;
            border = new java.awt.BorderLayout();
            container = getContentPane();
            color = new java.awt.Color (0xeeeeee);
            setTitle ("SpeedLogo Animation");
            setBackground (color);
            container.setLayout (border);
            sliderField = new java.composite.SliderFieldPanel();
            sliderField.setCurrentValue (10);
            logo = new java.animation.Logo();
            logo.startAnimation();
            container.add (logo, BorderLayout.CENTER);
            container.add (sliderField, BorderLayout.SOUTH);
            sliderField.addPropertyChangeListener
                (new sliderField_PropertyChange());
        }

        class sliderField_PropertyChange
            implements java.beans.PropertyChangeListener{
            public void propertyChange
                (java.beans.PropertyChangeEvent e) {
                int value;
                value = sliderField.getCurrentValue();
                logo.setAnimationRate(value);
            }
        }
}
```

Figure 7: The Corresponding Java Code for the Manipulation Operators in Figure 6

5 CONCLUSIONS

Our work makes the following contributions. It provides, to the best of our knowledge, the first attempt at the standardization of component compositions elevating compositions to first class citizenship status. We provide a CoCo composition model that is both flexible and extensible, allowing developers to extend the standard to include at a later time other composition styles of component-based development. The composition model is described using XML schema. At the lowest level, composition scripts can be described using the composition model implementation to describe an actual composition of two or more components. A composition script written in XML documents conforms to a specified com-

[1] The component instance `sliderField` allows any component instance implemening the interface `PropertyChangeListener` to register for the event `PropertyChange` while the component instance `logo` does not implement such interface.

position model specified by an XML Schema, which in turn conforms to the general guidelines of the composition model.

The composition model is analogous to defining the grammar of a composition language, while the composition script [2] provides the program that describes the actual composition of two or more components. The composition model thus provides an *extensible* composition grammar.

REFERENCES

Achermann, F. (2002). *Forms, Agents and Channels - Defining Composition Abstraction with Style*. PhD thesis, University of Bern, Institute of Computer Science and Applied Mathematics.

Achermann, F., Lumpe, M., Schneider, J.-G., and Nierstrasz, O. (2001). Piccola – a Small Composition Language. In Bowman, H. and Derrick, J., editors, *Formal Methods for Distributed Processing – A Survey of Object-Oriented Approaches*, pages 403–426. Cambridge University Press.

Akerley, J., Li, N., and Parlavecchia, A. (1999). *Programming with VisualAge for Java 2 (2nd Edition)*. Prentice Hall PTR.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249.

America, P. (1990). Designing an Object-Oriented Programming Language with Behavioural Subtyping. In *The REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK. Springer-Verlag.

Box, D. (1998). *Essential COM*. Addison-Wesley Publishing Company.

Butler, S. and Duke, R. (1998). Defining Composition Operators for Object Interaction. *Object Oriented Systems*, 5(1):1–16.

Chappell, D. (2002). *Understanding .NET: A Tutorial and Analysis*. Addison-Wesley Professional.

Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: Architectural Description of Component-Based Systems. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of Component-Based Systems*, pages 47–67. Cambridge University Press.

Gschwind, T., Oberleitner, J., and Jazayeri, M. (2002). Dynamic Component Extension to Support Cross-Platform Development. Technical Report TUV-1841-2002-19, Technische Universitt Wien.

Heineman, G. T. and Councill, W. T. (2001). *Component-based Software Engineering*. Addison-Wesley Publishing Company, Reading, Massachusetts.

[2] We use composition script to denote the description of the composition that results in a composite component or application.

Muller, H. and Davidson, M. (1996). JavaBeans Specification: Getting Listeners from JavaBeans. http://java.sun.com/products/javabeans.

Oberleitner, J., Gschwind, T., and Jazayeri, M. (2003). The Vienna Component Framework Enabling Composition Across Component Models. In *The 25th International Conference on Software Engineering*.

Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., and Burt, C. (2001). A Unified Approach for the Integration of Distributed Heterogeneous Software Components. In *The Monterey Workshop on Engineering Automation for Software Intensive System Integration*, pages 109–119.

Roth, B. (1998). An Introduction to Enterprise JavaBeans Technology. http://java.sun.com/products/ejb.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335.

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

Stearns, B. (2001). Using Forte for Java to Develop and Deploy Enterprise Beans. http://java.sun.com/developer/technicalArticles/WebServices/ffjweb/.

Tansalarak, N. and Claypool, K. T. (2004). CoCo: Composition Model and Composition Model Implementation. Technical Report 2004-006, Department of Computer Science, University of Massachusetts - Lowell. Available at http://www.cs.uml.edu/techrpts/reports.jsp.

Vallecillo, A., Hernandez, J., and Troya, J. (2000). Component Interoperability. Technical Report ITI-2000-37, Departmento de Lenguajes y Ciencias de la Computacion, University of Malaga. Available at http://www.lcc.uma.es/∼av/Publicaciones/00/Interoperability.pdf.

Yakimovich, D., Travassos, G. H., and Basili, V. R. A Classification of Software Components Incompatibilities for COTS Integration. http://sel.gsfc.nasa.gov/website/research/tech-study.htm.