On Component Composition Languages

Francisco Curbera, Sanjiva Weerawarana and Matthew J. Duftler
Component Systems Group
IBM T. J. Watson Research Center
Hawthorne, NY 10532, USA.
mailto:{curbera,sanjiva,duftler}@us.ibm.com

April 15, 2000

POSITION PAPER

Abstract

The benefits of software component composition are today widely accepted. However, component oriented software development is not yet as widespread as its multiple advantages may suggest. This is so in spite of the maturity reached by several component models (Microsoft’s COM, JavaBeans, OMG’s CORBA), and their general acceptance by large communities of developers. Thus, while components are being “used” in software development, the development process itself is not fully component-oriented. One major roadblock limiting the adoption of a component-oriented development process is the lack of viable component composition languages. There are, however, major costs associated with the adoption of a new language, and reducing adoption costs must be a major objective in the development of a component composition language.

In this paper we propose a list of requirements for a successful component composition language, in which we attempt to capture the needs of providing first class support for composition operations and assuring seamless integration into existing development environments. Next, we identify the problems associated with the design of a language meeting these requirements, and present a practical approach to deal with them.

Introduction

The benefits of software component composition are today widely accepted, see [5, 6, 16, 10, 2, 14], however, component oriented software development is not yet as widespread as its multiple advantages may suggest. This is so in spite of the maturity reached by several component models (Microsoft’s COM, JavaBeans, OMG’s CORBA), and their general acceptance by large communities of developers. Thus, while components are being “used” in software development, the development process itself is not fully component oriented. One major roadblock limiting the adoption of a component oriented development process is the lack of viable component composition languages. As has been argued in [12] and [16], component-oriented development is likely to be much more successful when first-class mechanisms enabling simple forms of composition are used.

Traditional programming languages are not the best suited for component composition. Since their syntaxes and semantics do not support component composition concepts in a first-class manner, composition operations are supported using other existing language elements like, say, method calls. As a result, the composition operations are lost amongst the rest of the code and the compositional structure is obscured. A discussion of the shortcomings of object oriented languages when applied to component composition can be found in [1]. Scripting languages suffer from these same shortcomings.
A solution is to introduce a component composition language, that is, a language in which the basic component composition operations are supported in a first class manner.

Extending the syntax and semantics of existing languages is one way to achieve this. It can be argued, though, that a special purpose composition language will do a better job capturing the specific nature of composition operations. Moreover, as noted in [1], object–oriented languages like Java and object oriented design tend to be used to produce domain specific designs, rather than standard architectures more suitable for the kind of reuse expected from software components.

The introduction of a new language has some major practical problems, though. It requires retraining and the development or adaptation of tools to support it. Furthermore, component models and run–time models to interact with other languages must be developed. Thus, an approach where a new language, a new component model and a new run–time is needed is not immediately suitable as a mechanism to enable component–oriented development in practice.

We believe that the quick acceptance of the component oriented methodology depends on providing an answer to this apparent contradiction. In other words, it depends on the availability of a component composition language that can be easily integrated in today’s application development environments. This paper discusses the requirements for this type of language.

Requirements

The problem we described in the Introduction has two parts: designing a composition language, and assuring its easy integration in development environments. In this section we propose a list of requirements for a component composition language that address both of them.

Several papers have dealt with the problem of specifying requirements for successful composition languages. The following discussion owes much to the ones found in [12], [13] and [3].

Our first requirement states a list of composition operations that a composition language should support.

1. The following composition operations must be supported by the language:

   - **Binding communication channels.** Communication channels let components exchange data and invoke behavior. Good examples are pipes and filters, and event notification in JavaBeans.

   - **Creating higher level component aggregates.** In this operation components are combined to produce higher order functional constructs. The combination typically involves creating a hierarchy of components, as when creating graphical user interfaces.

   - **Macro expansion** of parametrized components. Macro expansion can be used in several ways to compose components. In the COMPOST language, [3], source components are connected together by expanding (binding) “generalized program elements” present in each component’s code. Another case of composition by macro expansion is described in [4].

   - **Recursive component composition.** Component composition is used to create new components, rather than an application. This is a powerful technique that enables components to become software abstractions at different levels, and provides support for top-down progressive refinement design strategies. It is also has an important role providing scalability to the language, since it allows using the same language composition abstractions at different configuration levels.

   A language solely devoted to component composition must also provide effective separation of concerns between the person doing the composition and the developer of components. This is nothing but a restatement of the principle that the composition of components must require no knowledge of their implementations. In particular, the language must provide a way to address “compositional mismatches”, i.e. situations when the interfaces of two components are incompatible and don’t allow direct composition.

2. The language should allow the specification of “glue code” to deal with compositional mismatches.
Glue code provides the bridge through which the two interfaces can interact. In object oriented design this corresponds to the “adapter” pattern, [7].

The next requirement deals with the important issue of reusing component application designs.

3. The language should support component frameworks.

Here the notion of a component framework is similar to the frameworks found in object oriented design, see [7, 9] for instance. It is defined in [16] as a software architecture that provides basic relationships among components and allows instances of those components to be plugged in the framework. Frameworks are important tools that provide component assemblers with the infrastructure needed to build structured applications. Frameworks are also important as a knowledge sharing mechanism and as enablers of large scale component oriented development.

In order to assure seamless integration of the language into current development environments, we state in our requirements list the need for low adoption costs, and the ability to reach different development platforms as possible:

4. Reduce to a minimum the learning process for the language. In particular, use whenever possible existing languages, syntactic and semantic conventions. The Java language and the XML syntax would be good starting points according to this criterion.

5. Eliminate the need for new support tools, whenever possible. Existing development environments should be able to provide support for the language with minimal investment.

6. The language should support most common component models. In particular, the JavaBeans, COM and CORBA models should be supported.

The Bean Markup Language

The purpose of this section is to discuss the language design problem that is defined by the requirement list presented before. We do so by discussing how these requirements have been addressed in the design of the Bean Markup Language (BML).

BML is a declarative language for the composition of JavaBeans components. BML supports most major component composition operations in a first–class manner, and has a very low adoption cost both for programmers and machines. In this section we do not describe BML; we just focus on how BML addresses the requirements of the previous section. A detailed description of BML can be found in [18]. For a description of the JavaBeans component model see [8, 15]. A simple example of how BML is used to compose an application is included in figure 1.

XML Syntax

BML intentionally de–emphasizes the importance of syntax. From the two alternatives of choosing a syntax with multiple elements and structures (e.g., a Java–like syntax), or following a relatively “syntax–free” approach (e.g., the Lisp way), the second option was judged more likely to allow the language to satisfy requirements 4 and 5 from Section . This is the reason why XML was chosen as the meta language upon which to define the language syntax. Its XML syntax is in fact the main reason why BML complies with those two requirements.

XML is a meta language for defining new languages that conform to the syntactic model of XML [11]. XML is syntactically very simple, allowing very limited syntactic options. The essential syntactic choice is whether to use an XML attribute or an XML element to represent features of the language. XML, on the other hand, is already a widely embraced industry standard, its simple syntax is well known by many developers, and supporting middleware is available for all major computing platforms.
<xml version="1.0"/>
<script>
  <bean class="java.awt.Frame" id="frame">
    <property name="title" value="IBM Juggler"/>
    <event-binding name="window" filter="windowClosing">
      <script>
        <call-method target="class:java.lang.System" name="exit">
          <cast class="int" value="0"/>
        </call-method>
      </script>
    </event-binding>
    <add>
      <bean class="demos.juggler.Juggler" id="Juggler"/>
      <string value="Center"/>
    </add>
    <add>
      <bean class="java.awt.Button">
        <property name="label" value="Start"/>
        <event-binding name="action">
          <script>
            <call-method target="Juggler" name="start"/>
          </script>
        </event-binding>
      </bean>
      <string value="North"/>
    </add>
    <add>
      <bean class="java.awt.Button">
        <property name="label" value="Stop"/>
        <event-binding name="action">
          <script>
            <call-method target="Juggler" name="stop"/>
          </script>
        </event-binding>
      </bean>
      <string value="South"/>
    </add>
    <call-method name="pack"/>
    <call-method name="show"/>
  </bean>
  <script>
    <call-method target="Juggler" name="start"/>
  </script>
</bean>
</script>

Figure 1: The Juggler Script
While a Java–like syntax would have the advantage of providing a certain degree of familiarity to Java
developers, it would also have the disadvantage to being only Java–like, and not exactly Java. In fact, the
intended user of BML is the component composer, who may not even be a Java developer.

Binding Events

Binding communication channels is one of the main composition operations described in Section 3. In the
JavaBeans model inter–component composition channels are event streams. In order to do the binding, two
requirements must be met:

• The event source must be notified of the listeners’ interest in receiving the events.
• Event listeners must be of a suitable type which is statically defined by the event source.

BML uses the \(< event \ binding >\) element for this purpose, as in the next example:

\[
<bean class='java.awt.Button'>
  <bind-event name='action'>
    <bean class='MyActionListener' id='a1'/>
  </bind-event>
</bean>
\]

Notice that the component 'a1' must be of the appropriate type for this the binding operation to be valid.
This kind of binding of communication channels is hence fairly restrictive as the components must be statically
designed to be aware of each other’s event types. The following section discusses how this is generalized to
make event bindings more adaptable.

Writing “Glue” Code

Composing components that are not pre–designed to be linked together often requires the writing of “glue”
code to solve these compositional mismatches (recall requirement 2 from Section 3). In the JavaBeans case the
problem is even worse because the event binding architecture which requires that the event listener implement
a certain interface type.

BML addresses this requirement by allowing the component composer to author glue code in any of several
traditional scripting languages. The currently supported languages include JavaScript, Jacl, JPython and
VBScript.

This is an important design point. Traditional languages are better fit for writing glue code because,
typically, the glue code does not perform component composition, but rather some type of data adaptation to
allow components to interact. A composition language is clearly less suited for such tasks than a traditional
scripting language, except perhaps for the most elementary ones. Observe also that this further reinforces the
clearer separation between component authoring and component composition: while JavaBeans authors are
Java programmers, component composers need not be so.

The glue code is directly embedded in the composition script using a \(< \text{script} >\) element as the child of
an \(< \text{event} \ − \ \text{binding} >\). In lines 20 to 24 in Figure 1, for instance, a BML script is provided to cause the
invocation of the “start” method when an “action” event is received.

The code in these scripts is executed at run–time when events are generated by the event source component.
However, BML provides static scoping for the script, that is, any component that is referenced by the script
and was previously registered within its lexical scope will be available during script evaluation.

Aggregation

Aggregation of components into hierarchies is another major composition operation. BML supports it through
the \(< \text{add} >\) element. The following example illustrates the process of adding a java.awt.Button component
to a java.awt.Panel component:
The meaning of an aggregation operation is defined by the “container” into which aggregation is occurring. This is the default target bean (in XML terms, the parent <bean> element of the <add>), unless otherwise stated by the <add> element. BML’s approach is to stay away from differences in the semantics of the operation; only its compositional significance is of interest.

The mechanics of how the aggregation is implemented are part of BML’s assembly-time environment. This includes a registry (the adder registry) of code fragments (adders) that implement specific aggregation operations for specific container types. The separation of the compositional meaning of the operation from the mechanics of its implementation mechanism serves to further increase the declarative nature of BML: the component composer is only concerned with the desired aggregation structure and not with how that is to be actually realized.

Macro Expansion and Recursive Composition

BML has a form of macro expansion that supports treating existing BML scripts as defining pre-configured components, which can then be embedded and further configured on new scripts.

To achieve this BML allows using the name of a BML file as the value of the class name attribute in the <bean> element used to instantiate the component. The nested BML file is evaluated recursively and the resulting component is then used as the default target bean for further composition operations.

Consider this example:

```xml
<bean class='java.awt.Button'>
    <property name='background' value='0xff0000'/>
</bean>
```

In this example the first BML script takes the component produced by evaluating redbutton.bml and then sets its label property. The file redbutton.bml takes a Button component and sets its background color property to red and returns it. This simple example illustrates how a nested BML script can be used as defining a component which is then further configured and composed.

This approach amounts to macro expansion without parameterization. BML in fact allows parameterization of such scripts: the recursive invocation can be given arguments similar to how constructor arguments are given. The nested script can then retrieve the arguments and use them as it wishes. This allows the nested script to effectively be a template composition, with key parts filled in by the values of the parameters.

Note that this type of parameterized macro expansion does not enable true recursive composition because we can only manipulate the features of the returned component and not of an entire composition.

Beyond BML

The BML approach can successfully address most of the core component composition operations, and is designed to allow easy adoption in existing Java development environments. BML’s applicability in this space has
been validated by its successful use in several real-world applications as a composition language to describe composite applications [17].

Strong anecdotal evidence leads us to believe that BML’s choice of XML for a syntactic model and its support for arbitrary scripting languages for implementing “glue” code greatly enhanced BML’s applicability in the real world. The main limitation of BML’s composition approach is its inability to recursively compose more and more complex compositions. That is, BML is strong at composing applications, or final compositions, and not at composing other composable components.

There are several aspects to this problem. First is how the interface of the composition is to be defined. The interface of the composition consists of descriptions of its properties, events and methods. This problem is similar to that of designing an interface definition language and is straightforwardly solved. We have already designed and implemented an XML-based language for defining the interfaces of a composite component.

Once the composite component’s interface is defined, it must be implemented. There are two ways to do this: In one case, features of the components that comprise the composite component can be exposed as features of the composite component. For example, properties and events of beans used in the composition become properties and events of the composition. Our approach to enabling this mechanism is to extend the composition language to support “exporting” features of the beans used in the composition.

In the second case, the features of the composite component are not direct features of any of the components that form it. We call these “synthesized” properties and events. Typically, one would use scripting to provide the code supporting such synthesized properties and events. The language will also need to architect a way of “exporting” this logic as part of the composite’s interface.

Our current activities with BML are to extend it to support true recursive composition.

Composition Patterns and Frameworks

Once the composition language supports recursive composition, we can address how to build composition patterns and frameworks. A composition pattern is a composition with pre-specified hooks where specific component types can be inserted at a later time. Composition patterns are to component composition what design patterns are for object-oriented design. Once recursive composition and composition patterns are supported, component frameworks (which are in many ways patterns with a broader scope) can be built. We envision extending our basic XML-based approach with appropriate features to cover the basic case of application composition through the advanced case of component patterns and frameworks.

Conclusion

We have presented our vision of component composition languages in this position paper. We first identified a set of requirements of composition languages and then describe an approach for meeting these requirements that is directly applicable to the real world. Finally we considered how to evolve the application composition language we have developed to a truly recursive composition language supporting composition patterns and frameworks.

References


