USENIX Association

Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '01)

San Antonio, Texas, USA
January 29 - February 2, 2001

Bean Markup Language: A Composition Language for JavaBeans Components

Sanjiva Weerawarana, Francisco Curbera, Matthew J. Duftler,
David A. Epstein*, and Joseph Kesselman
Component Systems Group
IBM TJ Watson Research Center
Hawthorne, NY 10598
{sanjiva,curbera,duftler}@us.ibm.com, depstein@vastvideo.com, keshlam@us.ibm.com

Abstract

Although the benefits of software component composition are today widely accepted, 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. This paper introduces a component composition language specifically designed for the composition of JavaBeans components.

The Bean Markup Language (BML) supports component composition in a first–class manner. BML has language constructs for describing inter–component bindings, for constructing aggregates of components, for macro expansion and for implementing certain types of recursive compositions. Further, it allows the specification of "glue code" in any traditional scripting language (for example, JavaScript) to enable powerful adaptation during composition.

*Presently Vice President, Development & Architecture, VastVideo Inc., Astoria, NY 11106.

1 Introduction

The benefits of software component composition are today widely accepted, see [6, 7, 21, 11, 2, 15], 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 [13] and [21], component-oriented development is likely to be much more successful when first-class mechanisms enabling simple forms of composition are used.

Component–oriented development is a natural evolution of the object–oriented development paradigm. Components provide a programming abstraction in terms of events, properties and methods. The properties and methods of a component allow the component to be configured and events are how the component communicates interesting information to its consumers. In the component–oriented development model, application development becomes a matter of "scripting together" a set of such components, where the components themselves are sometimes bought from a collection of third parties and sometimes developed in–house. This type of composition enables loose coupling and provides the necessary hooks for adapting pre–built components as needed to form the desired aggregation. Object-oriented development is clearly the predominant methodology used in developing the

components themselves [21, 11].

The key technologies that enable component-oriented development are a component model and a composition mechanism. A component model is a set of conventions and a run-time architecture that provide an environment to define and manipulate software components. The definition of a software component varies. However, a common theme is that it is an executable, self-contained, dynamically loaded/bound module that exhibits certain types or interfaces or contracts to other components that adhere to the same component model [7]. The three most popular component models in use today are: Microsoft's COM [5], OMG's CORBA [18] and JavaSoft's JavaBeans [20]. The work described in this paper assumes the JavaBeans component model, but it could be implemented with any of these models.

Component composition is the key programming task required and enabled by component-oriented development [2]. Component composition is the process of creating component instances, configuring them and putting them together to form composite components or applications. Configuring components consists of manipulating their properties and also invoking their methods. Putting components together consists of describing component–to–component relationships as well as aggregations such as component hierarchies. An ideal language for component composition would have first-class syntax and semantics to support such composition operations.

Component composition can be performed in a variety of ways. The obvious way is to use a traditional programming language to write code that creates instances of components and composes them together by using the appropriate method calls. This task is commonly done by the "main" procedure of an application or that of a composite component.

Traditional programming languages are however 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].

A second common approach to software composition is the use of scripting languages. Scripting languages are programming languages which are supposed to be in some sense "easier" to program with: they are typically loosely typed and interpreted. They are commonly used for application prototyping, configuration, customization and extension [17, 11]. Scripting is a natural counterpart to component-oriented development - components can be written in standard object-oriented languages and then "glued together" to form applications. However, as a mechanism for component composition, scripting languages such as PERL [22], Tcl [16] and JavaScript [8] suffer from the same problem as traditional programming languages, their lack of first class support for composition. In fact, scripting languages do not add abstractions to programming; their primary goals are to reduce complexity by eliminating syntax and types to make programming "easier," not to change the level of abstraction.

Visual composition is another popular approach to component composition. A visual builder allows one to select components from a palette, place them on a composition editor and visually "wire" together the component interactions. Where required, additional behavior can be added using scripts, for instance, to intercept an event on its path from a source to a target and trigger special actions. The JavaBeans component model in fact recognizes the role of visual builders and provides for the component to distinguish between build-time and run-time. Using this, a JavaBeans component may, for example, present a build-time user interface that can be used to configure the component. Visual composition clearly provides first-class support for the composition operations described earlier. Visual composition's power is also its failing however: it is interactive and graphical, and, consequently, not an option in scenarios where the composition is done by a non-interactive mechanism. This is the case, for instance, when user interfaces are automatically generated from data input specifications. A first class representation of the composition would allow both generation methods (interactive and non-interactive) to interoperate, by acting as a neutral intermediate format. Moreover, if appropriately designed, the intermediate format could also be directly manipulated by developers.

Hence, a solution to the shortcomings of existing compositions techniques is to introduce a component composition language, that is, a language in

which the basic component composition operations is supported in a first class manner.

Extending the syntax and semantics of existing languages looks like an attractive option, but it can be argued that a special purpose composition language will do a better job capturing the specific nature of composition operations. Moreover, it is noted in [1] that 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 most relevant effort in the definition of a composition language is probably Piccola [1], a declarative composition language founded on a variant of the π calculus, in which components are viewed as interacting processes. Piccola is a very small language, which is able to support a variety of component models through the definition of different component composition ("architectural") styles. Piccola is an on-going research effort.

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 runtime 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.

Hence, the problem we are interested in can be formulated as follows:

How to design a component composition language which can be seamlessly integrated in today's software development environment.

We believe that an answer to this problem can be a key to successfully driving today's development methodology toward the component oriented development paradigm. This paper describes an answer to this problem, the Bean Markup Language (BML), a declarative language for the composition of JavaBeans components.

The rest of the paper is organized as follows. Section 2 states the requirements for a composition language that can capture the design problem stated above. Section 3 describes the design of the BML language and how BML addresses these requirements. Section 4 describes the BML implementation and runtime support. Finally, in Section 5 we address open problems and research issues.

2 Requirements for a Composition Language

The purpose of this section is to map the design problem stated in the Introduction to a list of design requirements. The starting problem has two parts: designing a composition language, and assuring its easy integration in development environments.

Several papers have dealt with the problem of specifying requirements for successful composition languages. The following discussion owes much to the ones found in [13], [14] 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] and [19] discusses a mechanism to maintain correct scoping while generating programs.

- **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 correspond to the "adapter" pattern, [9].

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 [9, 10] for instance. It is defined in [21] 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 primitives must allow easy extension to support alternative component models. While the focus of this work has been the JavaBeans model, it should enable a direct extension to support the COM and CORBA models.

3 BML: A Composition Language for JavaBeans

The BML language was designed to meet many of the requirements we have identified in the previous section. This section describes the BML language, its design principles, and some of its most relevant features.

This section is organized as follows: first we explain some of the general design decisions behind BML. Finally, we describe the major language elements and explain the support that BML provides for the composition of components and other relevant features of the language.

Design Principles

BML has been designed as an XML–based declarative language for describing the composition of JavaBeans applications. This statement summarizes three major design principles, which we review in this section.

XML syntax. BML intentionally de-emphasizes the importance of syntax. From the two alterna-

tives 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 2. This is the reason why XML was chosen as the syntactic model for the language. Its XML syntax is in fact the main reason why BML complies with those two requirements.

XML languages follow a relatively simple syntax model (see [12]), and are described using the XML DTD [24] or the XML Schema [25] metalanguages. The XML model allows very limited syntactic options, essentially the choice of 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.

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.

A declarative language. BML is designed to describe the composition of a set of components rather than to describe how the composition is to be implemented. To fully understand this distinction we state here the four phases of the component development process:

1. **Authoring–time**. Components are created, typically using an object oriented language, and packaged for use by third parties.
2. **Composition–time**. This is design–time, when components are selected, configured and the desired composition is described. The role of component languages is to capture this composition.
3. **Assembly–time**. Part of the application startup time. The composition described in the component language script is realized into an executable application, typically by a component language processor.
4. **Run–time** After the composition is performed, the application runs to perform its function. Non-compositional processing happens at this time, typically by executing the component's own code.

The role of BML is to represent the structure of a composed application as designed at composition time. The actual assembly of the components is the role of the language processor at assembly time. BML defines an assembly–time environment to support this distinction (the assembly–time environment is described later in this section and in Section 4). This is the reason why we describe BML a as declarative composition language. It must be remarked that, while BML allows the inclusion of sections of "glue code" for the purpose of solving compositional mismatches and "configuration instructions" for configuring individual components for composition, the BML language's compositional elements are declarative.

Application versus component composition. Compositions can be either "final" or reusable. Final compositions are *applications*. Reusable compositions are themselves components and can be used in new compositions, both final and reusable. The main difference between the two is that reusable compositions present a well defined public interface that identifies them as components in the component model under consideration, and allows reuse. BML is designed to enable the creation both types of compositions, by providing component definition language elements in addition to the basic compositional and configurational elements. The ability to define reusable components in the language provides support for the for recursive composition requirement listed in Section 2.

3.1 BML Language

As an XML based language, BML uses different XML elements for each composition operation. This section presents the BML solution to the composition language problem by describing how it addresses each of the requirements listed earlier. We describe only the essential features of each element in this document; complete documentation can be found in the BML User's Guide, which is part of the BML distribution [23]. The syntax of BML is summarized in a BNF–like form in Table 1.

The role of BML is to capture a composition. A composition script is represented in BML by a <script> element. The contents of this element are

<script>	::=	(S | <cast>)+
<bean>	::=	<args>? S*
<args>	::=	V+
<property>	::=	V?
<field>	::=	V?
<event-binding>	::=	<bean> | <script>
<call-method>	::=	V*
<cast>	::=	V?
<string>	::=	text
<add>	::=	V+
C	::=	(<bean> | <string> | <property> | <field> | <call-method> | <script>)
S	::=	(C | <event-binding> |
	::=	<add>)
V	::=	(C | <cast>)

Table 1: BML Syntax Summary

arbitrary BML elements and the result of evaluating it is the value of evaluating the last child element.

We start the description of the BML language with a small, yet complete example.

The Juggler

This section provides an simple example of a BML application. The purpose is to give the reader an early view of a significant subset of BML.

The example shows how BML can be used to compose a collection of AWT components into an application. The application includes an animation component and two buttons that control it, as well as a window frame component that acts as a container component. Figure 1 shows the resulting application. The example code is shown in Figure 2.

We now briefly explain how the code in in Figure 2 works.

Note that line 0 is the XML declaration which is required of XML documents. In line 2 a new a new script of BML statements is started with the <script> element. The <bean> element in line 3 creates a component of type java.awt.Frame and uses the *id* attribute to assign to it the name "frame". In line 4 the title property of the frame bean is set. The <event-binding> element in line

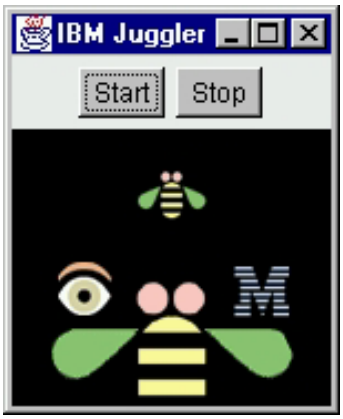

Figure 1: The Juggler Application

5 binds the script in lines 6–10 so it is run when a "window" event occurs and the event is delivered via the windowClosing method. The script contains one statement (lines 7–9) which causes the program to exit.

On line 14 the animator component is created and given the name "Juggler". Lines 13–16 aggregate this component into the container "frame" at the center position using the <add> element. On line 18 a button component is created, its label property is set to "Start" on line 19. Line 20 binds the script on lines 21–23 to be run when an "action" event occurs on the button. The script invokes the start method of the "Juggler" component using the <call-method> element. Observe that the target component is identified using the "target" attribute. Lines 17–27 aggregate the button component into the container "frame" in the north position. Similarly, lines 28-38 create another button component (this one for stopping the animation) and aggregates it to the "frame" component. Lines 40-41 invoke the "pack" and "show" methods of the frame in order to bring it to the screen. Finally, line 43 invokes the "start" method of the animator component to initiate the animation.

In the following sections we review in detail some important aspects of the language.

Naming and Scoping in BML

A mechanism to identify components is fundamental to any composition language. In the previous section we have seen how beans can be created with the

```
0     <?xml version="1.0"?>
1
2     <script>
3       <bean class="java.awt.Frame" id="frame">
4         <property name="title" value="IBM Juggler"/>
5         <event-binding name="window" filter="windowClosing">
6           <script>
7             <call-method target="class:java.lang.System" name="exit">
8               <cast class="int" value="0"/>
9             </call-method>
10          </script>
11        </event-binding>
12
13        <add>
14          <bean class="demos.juggler.Juggler" id="Juggler"/>
15          <string value="Center"/>
16        </add>
17        <add>
18          <bean class="java.awt.Button">
19            <property name="label" value="Start"/>
20            <event-binding name="action">
21              <script>
22                <call-method target="Juggler" name="start"/>
23              </script>
24            </event-binding>
25          </bean>
26          <string value="North"/>
27        </add>
28        <add>
29          <bean class="java.awt.Button">
30            <property name="label" value="Stop"/>
31            <event-binding name="action">
32              <script>
33                <call-method target="Juggler" name="stop"/>
34              </script>
35            </event-binding>
36          </bean>
37          <string value="South"/>
38        </add>
39
40        <call-method name="pack"/>
41        <call-method name="show"/>
42      </bean>
43      <call-method target="Juggler" name="start"/>
44    </script>
```

Figure 2: The Juggler Script

<bean> element, named using the *id* attribute, and located with the *target* attribute. Assigning names to new components is optional, but if a name is assigned then the component is registered in current scope with that name.

BML is a lexically scoped language. A scope is defined by the <script> and the <bean> elements. The default scope is created by the opening <script> element and nested scopes are explicitly created by nesting <script> elements; nested <bean> elements implicitly create a new scope. The scoping semantics are as usual with inside–out visibility and standard shadowing rules. The scope is represented at assembly–time as an "object registry", a registry which provides a name–to–reference mapping and is part of the BML language processor's environment during assembly–time.

A related issue is how BML uses XML's containment model to effect a Pascal–style "with" operator. Recall from our previous example that the *target* attribute is used to name the bean on which an operation is to be performed. In BML the default target for a composition operation is the closest enclosing component, as in the next example.

```
<bean class='java.awt.Button'>
  <property name='label'
            value='Click Me'/>
</bean>
```

Configuration of Components and Type Conversion

JavaBeans components may have configurable properties. BML uses the <property> element for this purpose. The example in Figure 1 shows that the value of the property can be encoded using the *value* attribute, as long as this value can be represented as a string.

When there is no possible string representation, the <property> element is given one child element, the result of evaluating which becomes the value assigned to the property. The following example shows how one can change the layoutManager property of a Panel component:

```
<bean class='java.awt.Panel'>
  <property name='layoutManager'>
    <bean
      class='java.awt.BorderLayout'/>
  </property>
</bean>
```

The <property> element also supports retrieving property values. This is effected by omitting any value for the property; thus, if a value to assign the property is not found, it is treated as an "rvalue" instead of as an "lvalue."

We now discuss type conversions issues arising when property values are encoded as strings. We must note first that this issue is a consequence of the lack of typing information in XML, which results in all the data being encoded as strings.

If the type of the property being set is not a string, a type conversion must be performed before the value can be set. For instance, one may wish to set color–valued properties by giving a string containing the RGB representation of the color. BML's approach is to separate the type conversion problems from the compositional problems as much as possible. All the type conversion logic is considered part of the assembly–time environment of the language, and is not reflected in the BML script.

The BML processor's assembly–time environment contains a registry, called the type converter registry, which is a collection of code that is able to convert data from one type to another. If a type conversion is deemed necessary, the processor will transparently invoke the appropriate converter in order to effect the setting of the property. The type converter registry mechanism serves to improve the declarative nature of BML: it enables one to concentrate on the required composition operations and defer the issues of *how* to realize them until later (and probably to someone else).

Type conversions can also be explicitly requested in BML. The <cast> element is a utility element used to explicitly request a type cast, or to explicitly invoke a type converter to change the type of a value. An example of is provided in line 8 of Figure 1, where a string to integer conversion is requested. Explicit casts are commonly found in BML in the the arguments of a <call-method> invocation (lines 7 to 9 in Figure 1), a common way of performing more complex configuration of components.

Binding Events

Binding communication channels is one of the main composition operations described in Section 2. 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'>
  <event-binding name='action'>
    <bean class='MyActionListener'
          id='a1'/>
  </event-binding>
</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 2). 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 <script> element as the child of an <event-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. Section 4 describes the implementation of <script>.

Aggregation

Aggregation of components into hierarchies is another major composition operation. BML supports it through the <add> element. The following example illustrates the process of adding a java.awt.Button component to a java.awt.Panel component:

```
<bean class='java.awt.Panel'>
  <add>
    <bean class='java.awt.Button'/>
  </add>
</bean>
```

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. Observe that the operation of nesting a <bean> element inside another has very

different semantics from the aggregation operation, since the first one corresponds only to the declaration of a bean inside the parent's scope.

Aggregations defined by different containers may require different data to be specified before the operation can be performed. In the above example, the <add> element has only one child because the layout manager that the panel uses (java.awt.FlowLayout) does not require any other information. However, the add operations included the example from Figure 1 take two arguments, since the default layout manager for the java.awt.Frame component, the BorderLayout, requires that we indicate the layout area in which the a component is to be added. In general, the first child element of <add> is expected to identify *what* to add and any other children are expected to be additional information as needed by the container's semantics for aggregation

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.

Recursive Composition

Recursive composition of JavaBeans requires a way to define a new component in terms of compositions of beans. The language elements presented so far deal with the connection of already defined beans, and are typically contained inside a <script> element. When this element is the root of the XML document, the BML script corresponds to a final application, that is, cannot be reused as a component (it can be reused through macro expansion as we explain later).

Recursive composition requires additional language support to define the constructor, properties, methods and events of the new bean using compositions of beans. This section describes the creation of new JavaBeans with BML. In the next section we present the related function of macro expansion in BML, which is way of reusing preconfigured BML scripts.

The example in Figure 2 shows how the Juggler application of Figure 1 can be wrapped in a bean. In this particular case, almost all the code from Figure 1 has been included as the constructor of the new bean, while two method calls have been exposed as methods of the composition.

A new component type is defined in BML using the <beanDef> element. The class for the new component is derived from the *name* attribute. The constructor, properties, methods and events of the component are defined using the <constructorDef>, <propertyDef>, <methodDef>, and <eventDef> elements respectively. These definitions can in general be provided in two ways: by *delegation* or by direct *implementation*.

When delegation is used, the composite's property, method or event is mapped to a property, method or event of a bean which is part of the composition. For example, in lines 59 to 66 of Figure 3 methods, properties and events of the composite bean are define by delegating to the $frame\ Juggler$, *start* and *stop* beans. In all these cases, the *name* attribute gives the name of the new method, property or event in the composite, the $sourceBean$ attribute is used to identify the delegation bean, and *method*, *property* and *event* identify the method, property and event in the source bean.

When using direct implementation, the implementation is specified by a nested <script> element containing a regular BML script (which includes no bean definition elements). An example of this is provided by the constructor in Figure 3, which includes most of the code from Figure 2. Constructors are defined using a <constructorDef> element, and can only be defined by direct implementation, never by delegation.

Constructors are different from other bean elements in another important aspect. The naming scope defined by the (top level) <script> element in a constructor's definition is considered global for the complete bean definition. That is, the identifiers introduced in this script are visible everywhere in the bean. This is used, for instance, in the identification of the beans used in all definitions by delegation. In the examples from figure 2, the names specified by all the $sourceBean$ attributes correspond to beans that were registered in the constructor's script. The identifiers of beans defined in the implementation of

```
<?xml version="1.0"?>

<beanDef name="CompositeJuggler">
  <constructorDef>
    <script language="bml">
      <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" id="start">
            <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" id="stop">
            <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"/>
      </bean>
      <call-method target="Juggler" name="start"/>
    </script>
  </constructorDef>

  <methodDef name="show" sourceBean="frame" method="show"/>
  <methodDef name="start" sourceBean="Juggler" method="start"/>
  <methodDef name="stop" sourceBean="Juggler" method="stop"/>

  <propertyDef name="startLabel" sourceBean="start" property="label"/>
  <propertyDef name="stopLabel" sourceBean="stop" property="label"/>

  <eventDef name="window" sourceBean="frame" event="window"/>

</beanDef>
```

Figure 3: The Juggler Bean

methods, properties of events are not visible outside their own script block.

Macro Expansion

BML provides a form of macro expansion that allows reusing existing BML scripts, 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 within a new scope of its own, and the resulting bean is then used as the default target bean for further composition operations.

Consider this example:

```
<bean class='redbutton.bml'>
  <property name='label'
            value='Red Button'/>
  ...
</bean>
```

where redbutton.bml is:

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

In this example the first BML script takes the bean 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.

This type of parameterized macro expansion is not true recursive composition because we can only manipulate the features of the returned component and not of an entire composition. See the previous section for a description of how recursive composition is supported in BML.

4 Implementing BML

We have implemented two BML language processors: an interpreter and a compiler. Both implementations are designed to be embeddable and provide full access to the assembly–time environment as well as to the run–time environment. The assembly–time environment has been pre–populated with a collection of type converters and adders that provide commonly used type conversions and aggregation capabilities, respectively. These can be augmented by the host of the BML processor by accessing the environment and adding new capabilities to the registry.

The interpreter receives the BML document as an XML tree and functions based on whether the outermost element is a $beanDef$ element or a $bean/<script$ element. In the latter case, it uses Java reflection to implement the composition operations. In the case of $beanDef$, the interpreter will use the compiler to compile the bean definition upon first use and then reuse the resulting class.

The BML interpreter implements static scoping by performing name registration and resolution against the object registry in scope. Each $< script >$ element introduces a new scope by creating a new registry which cascades upwards to the scope that embeds it. For event handler scripts ("glue" code), which are scripts whose execution is deferred until run–time, static scoping is achieved by storing the statically scoped registry with the script to be run at run–time. For example, the event script in line 22 of the example shown in Figure 2 binds statically to the registry in scope at assembly–time and then at run–time uses it to locate the "Juggler" component.

The compiler receives the BML script as an XML tree and uses Java reflection to generate the appropriate Java source code to implement the composition operations. If the outermost element is not $beanDef$, the compiler places the resulting code in a `main()` method. Otherwise, the bean definition

guides the target code generated.

The previously identified "assembly–time" phase for such generated composition code hence occurs at the startup of the execution of the generated code. The compiler allows one to generate code that is independent of the BML environment at assembly–time. That is, it can resolve type converters and adders as well as scoping at compile time if possible so that the generated code is straight Java code. If one wishes to have full embeddability of the generated code with the BML assembly–time environment, then it is necessary to generate code which BML dependent.

Implementing Event Bindings to Scripts

BML supports writing arbitrary scripts to be run as "glue" code. This is supported for any type of event thrown by any bean and is implemented with event adaptors, event processors and the event adaptor registry. The model consists of an event specific adaptor that receives the event from the source, delegates it to a generic event processor which then runs the script. This approach is a decomposition of the standard JavaBeans event binding model to allow dynamic look up and/or generation of event adapters.

Event adapters must implement a simple interface that is capable of receiving a handle to an event processor. An event adapter must be implemented and available from the event adapter registry for each event listener type. When the BML processor creates an event adapters and adds it as a listener to an event source, it tells the adapter what event processor to delegate the event to. Event processors are the entry point to the BML runtime and are responsible for delivering the event to the intended recipient script.

When an event adapter receives an event from an event source, it delegates the event to its event processor. The interpreter uses an event processor that actually delivers the event to a script and runs the script. The compiler can generate both customized event processors that perform this task, and customized event adapters that bypass the event processor mechanism entirely and directly deliver the event to the user's script.

The event adapter registry provides registration and lookup service for event adapters. We have also implemented the ability to on–demand generate event adapters in Java bytecode form. This eliminates the need to hand–write event adapters in many cases. (It is not possible sometimes because of security constraints of the runtime location; loading dynamically constructed classes is not always permitted.)

5 Future Work

Support Other Component Models

An important objective of the BML project is to achieve wide acceptance in the software development community by supporting other component models.

The challenge is to develop a a common component composition language supporting composition operations for the three major component models, JavaBeans, COM, and CORBA, in such a way that it can work with components from different component models in the same application. This work clearly depends on the availability of run–time bridges to go between the different models. Some of those already exist.

Concurrency Support

BML has not dealt with the issue of object concurrency. If components can be objects, and there is the possibility of concurrent execution, concurrency control can become a major issue. This is particularly important when communication channels are event streams, as in the JavaBEans model, since event handling is a common cause of race conditions and deadlocks in multithreaded environments (see [20]).

The question is whether, in these circumstances, the composition language should assure correct synchronization among components. When components from third party authors are used (maybe from several of them) and concurrent execution is necessary, it may become impossible to predict correct synchronization of the composition, and a positive answer would seem appropriate. This is the view expressed in [13], which underlies the design of the Piccola language.

The issue for BML is whether it is possible to integrate concurrency control while still keep the simplicity and transparency of the language. We have no answer to this yet.

6 Conclusions

We have presented an alternate approach to component composition in the form of a new composition language for the composition of JavaBeans components, the Bean Markup Language (BML). BML is a declarative language that uses the XML syntax to reduce the adoption barrier of both developers and machines. The language constructs are few and simple, and are designed to capture in a first–class manner the semantics of component composition. In spite its simplicity, BML provides support for most major composition operations. In particular, BML supports recursive composition, which assures scalability and enables top–down design methodologies.

BML allows composers to author event filtering scripts in arbitrary scripting languages, which opens up JavaBeans component composition to non–Java programmers. That is an important points that reinforces the notion that is not necessarily a programmer's job.

Still, BML does not address some relevant issues. One is concurrency control, which can be a critical issue in multiprocess environments. Finally, the objective of using BML as a vehicle to extend component oriented development equires that other component models be supported, if possible through a common composition language.

References

[1] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. Piccola - a small composition language. In H. Bowman and J. D. (Eds.), editors, *Formal Methods for Distributed Processing, an Object Oriented Approach*. Cambridge University Press, 2000, to appear.

[2] M. Aoyama. New age of software development: How component-based software engineering changes the way of software development. In *Proc. 1998 International Workshop on Component-Based Software Engineering*, 1998. http://www.sei.cmu.edu/cbs/icseworkshop.htm.

[3] U. Assmann. *Invasive Software Composition with Program Transformation*. Forthcoming habilitation, 2000.

[4] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The genvoca model of software-system generators. *IEEE Softw.*, 11(5), Sept. 1994.

[5] D. Box. *Essential COM*. Addison-Wesley, Reading, M, 1998.

[6] A. Brown. From component infrastructure to component-based development. In *Proc. 1998 International Workshop on Component-Based Software Engineering*, 1998. http://www.sei.cmu.edu/cbs/icseworkshop.htm.

[7] A. Brown and K. C. Wallnau. The current state of cbse. *IEEE Software*, September 1998.

[8] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilley, Cambridge, MA, 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, Reading, MA, 1995.

[10] IBM Corporation. *Building Object-Oriented Frameworks*. http://www.ibm.com/java/education/oobuilding.

[11] K. L. Kroeker. Software [r]evolution: A roundtable. *IEEE Computer*, 32(5), 1999.

[12] R. Light. *Presenting XML*. Sams.Net, Indianapolis, IN, 1997.

[13] O. Nierstrasz and T. D. Meijler. Requirements for a composition language. In O. N. Paolo Ciancarini and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, Lecture Notes in Computer Science 924, Berlin, 1995. Springer.

[14] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2), June 1995.

[15] O. Nierstrasz and D. Tsichritzis. *Object Oriented Software Composition*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[16] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, MA, 1994.

[17] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3), 1998.

[18] J. Siegel. *CORBA: Fundamentals and Programming.* John Wiley, New York, 1996.

[19] Y. Smaragdakis and D. Batory. Scoping constructs for software generators. In *Proc. First Symposium on Generative and Component-Based Software Engineering*, September 1999.

[20] Sun Microsystems. *JavaBeans*, 1997.

[21] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Harlow, England, 1998.

[22] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming PERL.* O'Reilley, Cambridge, MA, 1996.

[23] S. Weerawarana and M. Duftler. Bml v2.3 user's guide. *Available as a part of BML v2.3*, 1999. http://www.alphaWorks.ibm.com/formula/bml.

[24] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[25] World Wide Web Consortium. *XML Schema Part 1: Structures*, October 2000. http://www.w3.org/TR/xmlschema-1.