Using Wrappers to Add Run-Time Verification Capability to Java Beans

Vladimir Glina
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
vglina@vt.edu

Stephen Edwards
Department of Computer Science
Virginia Tech
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061, USA
edwards@cs.vt.edu

ABSTRACT
Because of limited information exchange between component providers and users, both these parties should perform component verification. Java Modeling Language, a notation which allows writing of behavioral specifications for Java programs, can be used for verification purposes. This paper shows that placing JML specifications in separate wrappers distributed in the binary form alongside components gives component buyers an additional value. The wrapper can serve for Java components verification on the user's side, verification checks can be enabled and disabled on per-class or per-package basis at run-time, and there is no performance overhead when they are disabled, unlike the traditional variant when checking code generated from JML specifications is placed directly into the underlying class bytecode. The paper describes wrapper design for Java Beans run-time verification and discusses advantages and challenges of it.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification—*programming by contract, assertion checkers, class invariants;* F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, invariants, assertions;* D.2.3 [Software Engineering]: Coding Tools and Techniques—*object-oriented programming;* D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids;* D.3.2 [Programming Languages]: Language Classifications – *design languages, constraint and logic languages*

General Terms
Languages, Verification.

Keywords
JML, Java Beans, run-time verification, design by contract, events handling

1. INTRODUCTION
Construction of software from commercial off-the-shelf (COTS) components is getting more and more popular. But limited information exchange between component providers and component users is a serious problem for it. Because of that, both component provider and user must perform component verification [1].

Assertion checking is an effective means to improve quality of software verification [2]. One of assertion checking tools is Java Modeling Language (JML), a notation for formal specification of behavior and interfaces of Java classes and methods. JML implements the Design by Contract (DBC) software development principle [3]: JML specifications (pre-conditions, post-conditions, and class invariants) placed in special comments within Java source code are transformed by the JML compiler into run-time checks. Originally that checks were placed directly into Java program bytecode. It created substantial performance overhead, so checks were removed before shipping software and did little for customers.

This paper discusses using JML-based assertion checking wrappers for verification and specification of Java Beans. The paper states that for component users this approach adds value to components by providing the following:

- checking the quality of connections between components;
- saving component user's time on producing tests;
- distribution checks in binary form alongside beans so that checks can be included without access to source code;
- run-time enabling or disabling checks on per-class or per-package basis;
- avoiding performance overhead when checks are excluded.

2. ASSERTION CHECKING WRAPPER IMPLEMENTATION
Assertion checking wrapper design for Java classes was proposed in [4]. The main idea of wrapper design for Java Beans is the same. Checking code is moved to a separate class, so that there are two classes extending the same interface: an unwrapped original class and the wrapper class which only provides assertion checking and calls methods of the unwrapped class to realize all other functionality. Objects of either class are created by a class factory, which allows separating the decision which class to instantiate from the object requesting the instance. Users can

```
public class MyBean implements PropertyChangeListener {
  protected /*@ spec_public @*/ int NonNegativeValue;
  //@ requires newValue >= 0;
  public void setNonNegativeValue( int newValue ){
    // implementation goes here …
  }
  // …
}
```

Figure 1. A fragment of the `MyBean` bean source code

enable or disable assertions on per-class basis, at run-time.

Wrapper-based design uses a customized version of the JML compiler `jmlc` which automatically generates four class files from the original source code shown in Figure 1. These classes, as Figure 2 shows, are:

- the implementation class providing original functionality (`MyBeanImplementation`);
- the wrapper class containing assertion checks (`MyBeanWrapper`);
- an interface that both the classes mentioned above implement (`MyBean`);
- a factory class (`MyBeanFactory`).

Figure 3 shows how inheritance is addressed. If `MyBean` inherits from `GeneralBean`, all the generated classes related to `MyBean` inherit from the corresponding classes related to `GeneralBean`. It means that if a class has JML specifications, all its superclasses are to be transformed by the JML compiler regardless of whether they have specifications or not.

As Figure 4 shows, the interface just re-declares all the public methods of the original bean class.

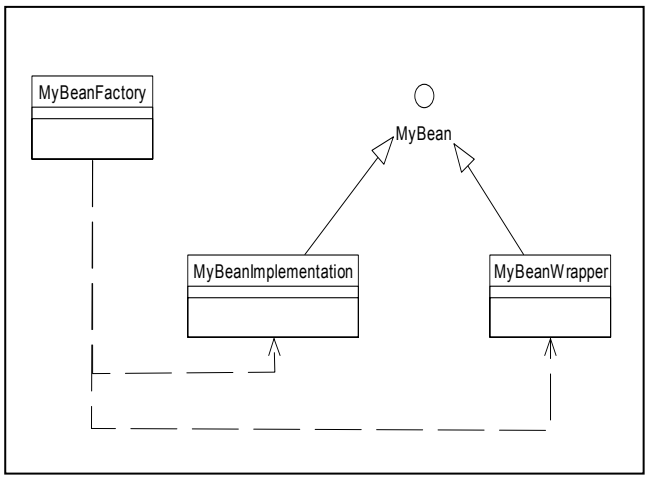

Figure 2. Transformation of the original class

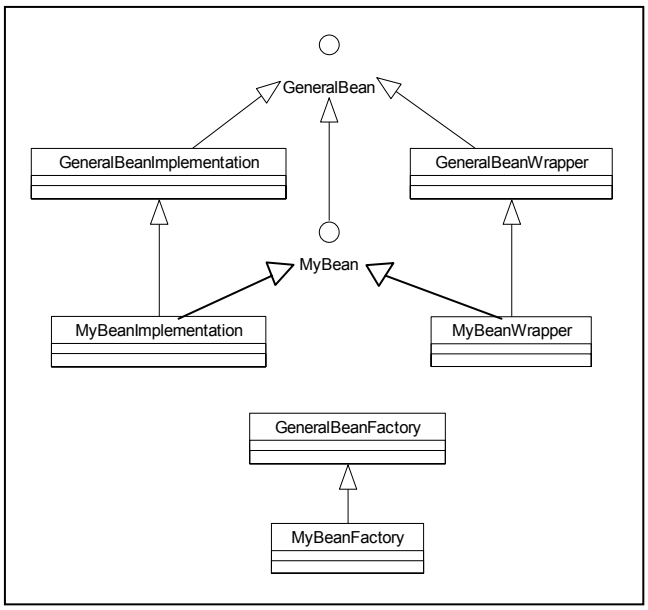

Figure 3. Inheritance: `MyBean` inherits from `GeneralBean`

In Figure 5, the wrapper corresponding to our sample bean is shown. The wrapper class masks the implementation class and adds assertion checks before and after every method. To do that, it holds the reference to the wrapped instance of the implementation class in the `wrappedObject` field. All the methods of the original class have corresponding methods in the wrapper class. Being called, the wrapper class methods at first perform pre-condition checks, then call the corresponding methods of the implementation class to perform core behavior, and after that make post-condition checks. The `isEnabled` object defines whether to perform particular checks at run-time. There is one such an object for every wrappable class and one similar object for each Java package. Thanks to these objects, it is possible to activate and deactivate run-time verification on per-class or per-package basis without access to source code, using a graphic control panel displaying the tree which maps the Java package nesting structure.

Figure 6 shows the factory class code. For every constructor in the original class, there is a corresponding factory method in the factory class. The factory queries its `isEnabled` field to decide what version of the object, wrapped or unwrapped, to create.

The implementation class has the same code as the original class

```
public interface MyBean {
  public void setNonNegativeValue( int newValue );
  // …
}
```

Figure 4. The interface that both wrapped and unwrapped classes implement

```
public class MyBeanWrapper implements MyBean {
   MyBeanImplementation wrappedObject;
   public static CheckingPrefs isEnabled = null;
   public void setNonNegativeValue( int newValue ) {
      if( isEnabled.precondition() ) {
        // the actual precondition check
        checkPre$setNonNegativeValue$MyBean( newValue );
      }
      wrappedObject.setNonNegativeValue( newValue );
   }
   // …                                                    }
```

Figure 5. The wrapper class

shown in Figure 1 except it gets another name:

```
public class MyBeanImplementation {
  // implementation goes here…
}
```

When a method returns an object, the object becomes wrapped when created.

A method of a wrapped object can have an exceptional postcondition for type *T* which describes what must hold true for the method to throw an exception of type of *T* (or a subtype of *T*). After a wrapped object method throws an exception, the wrapper checks if the corresponding exceptional postcondition is present and observed, and then the exception is rethrown. Otherwise, the assertion checking fails.

Non-public method calls and the same class method calls are checked in the same way as in the case of checking wrappers for regular Java classes [4].

To enable using wrapped object with existing code which does not use assertion checking and is probably available in the binary form only, the authors of [4] are implementing a custom class loader that can transform bytecode at load time if checking wrappers are used.

```
public class MyBeanFactory {
   public static CheckingPrefs isEnabled = null;
   public static MyBean instantiate() {
      MyBean result = new MyBeanImplementation();
      if ( isEnabled != null && isEnabled.wrap() ) {
        result = new MyBeanWrapper( result, isEnabled );
      }
      return result;
  }
}
```

Figure 6. The factory class

```
public static Object instantiate( ClassLoader cls, String
beanName, BeanContext beanContext, AppletInitializer
initializer) throws java.io.IOException,
ClassNotFoundException {
   // …
   if( result == null ) {
     // No serialized object, try just instantiating the class
     Class cl;
     try {
       if( cls == null ) {
         cl = Class.forName( beanName );
       }
       else {
         cl = cls.loadClass(beanName);
       }
     }
     catch (ClassNotFoundException ex) {
       if (serex != null) {
         throw serex;
       }
       throw ex;
     }
   }
}
```

Figure 7. Original `instantiate` method

Assertion checking wrappers for Java Beans require less change in coding practices in comparison with the ones for regular Java classes. In the latter case, developers have to get used to accessing attributes of classes through getters and setters that are added into the interface besides the methods defined in the original class, whereas properties of Beans can not be accessed other than through accessors. Also, all Java Beans are usually accessed through interfaces, not concrete classes.

Nevertheless, certain changes in Java Beans run-time environment, as well as in coding practice, are required for assertion checking wrappers implementation. Typically, a user can instantiate a bean either by using operator `new`, or by calling one of `the java.beans.Beans.instantiate` methods. The latter variant is equivalent to call of the method

```
java.beans.Beans.instantiate(     ClassLoader
cls,     String     BeanName,     BeanContext
BeanContext,  AppletInitializer  initializer
),
```

a fragment of which is shown in Figure 7, with some (or none) arguments set to null. After wrapper design implementation, an attempt to instantiate a bean using `new` will result in a compile-time error because the name of the unwrapped bean class is now belongs to the interface. The implementation of the `instantiate` method itself is to be changed in the place responsible for bean instantiation when there is no serialized object. The changes are shown in bold in Figure 8.

```
public static Object instantiate( ClassLoader cls, String
beanName, BeanContext beanContext, AppletInitializer
initializer) throws java.io.IOException,
ClassNotFoundException {
  //…
  if( result == null ) {
    Class cl;
    String factoryName = beanName.concat( "Factory" );
    try {
      if( cls == null  ) {
        cl = Class.forName( factoryName );
      }
      else {
        cl = cls.loadClass( factoryName );
      }
      Method instantiation = cl.getMethod( "instantiate",
                                           null );
      result = instantiation.invoke( cl, null );
    }
    catch( ClassNotFoundException ex ) {
      if (serex != null) {
        throw serex;
      }
      throw ex;
    }
}
```

Figure 8. The modified version of the instantiate method

```
public class MyBean implements PropertyChangeListener {
protected /*@ spec_public @*/ int NonNegativeValue;
  // …
  /*@   requires evt.getPropertyName() == “isPowerOn”;
    @   ensures  NonNegativeValue == 0;
    @ also
    @   requires evt.getPropertyName() == “numberOfTicks”;
    @   ensures  NonNegativeValue % 2 == 1;
    @*/
  public void propertyChange( PropertyChangeEvent evt )     {
     int n;
     if( evt.getPropertyName().equals(“isPowerOn” ) {
       n = 0;
     }
     else {
       if( evt.getPropertyName().equals(“numberOfTicks” ) {
         n = 4 * n + 1;
       }
     }
     setNonNegativeValue( n );
  }
}
```

Figure 9. Specification of the Bean event handling

3. TOWARDS SPECIFICATION OF JAVABEANS BEHAVIOR

The most substantial problem on the way of using design-by-contract tools for software components specification is dealing with concurrency, callbacks, and event handling. In this paper, we will describe how to use JML to check contracts on events a Java Bean is registered for.

As an example, let us suppose that there is a bean called `TickGenerator` which models a generator of electric impulses. For us it is enough to know this bean has the following properties: `IsPowerOn` of type `boolean` and `NumberOfTicks` of type `int`. When `IsPowerOn == true`, `NumberOfTicks` periodically increases. A bean user can revert the `IsPowerOn` value by clicking the corresponding button on the bean graphical interface. `IsPowerOn` and `NumberOfTicks` are bounded properties. The `MyBean` bean we dealt with at the beginning of the paper has registered itself with `TickGenerator` to be notified about changes of the properties values. The events handling logic of `MyBean` and the corresponding JML specifications are shown in Figure 9.

Of course, `MyBean` has very artificial events handling that is easy to describe. Nevertheless, it shows that JML specifications can be of benefit for events handling verification. The future work includes specifying event broadcasting, interaction of a group of components where one of them is listening for others, and beans serving in complicated environments that are hard to formalize (for instance, beans working with the FTP protocol).

4. ACKNOWLEDGMENTS

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

5. REFERENCES

[1] Beydeda, S., and Gruhn, V. The Self-Testing COTS components (STECC) Strategy – a new form of improving component testability. *Proceedings of the 29th Euromicro Conference(EUROMICRO'03)* (Belek-Antalya, Turkey, September 1-6, 2003). IEEE Computer Society, Los Alamitos, CA, 2003, 107 – 114.

[2] J.M.Voas. Quality time: How assertions can increase test effectiveness. *IEEE Software,* 14, 2 (Feb. 1997), 118-119.

[3] G.Leavens, Y.Cheon. Design by Contract with JML. *ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf*, draft, 2004.

[4] Tan, R., Edwards, S., "An Assertion Checking Wrapper Design for Java", *Proceedings of the Specification and Verification of Component-Based Systems workshop (SAVCBS'03)*, (Helsinki, Finland, September 1-2, 2003). Technical Report #03-11, Dept. of Computer Science, Iowa State University Ames, IA, 2003, 29-34.