Electronic Notes in Theoretical Computer Science 70 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 25 pages

jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java

Parker Abercrombie [a,1] Murat Karaorman [b,2]

[a] *College of Creative Studies*
University of California
Santa Barbara, Ca 93106

[b] *Texas Instruments, Inc.*
315 Bollay Drive, Santa Barbara, Ca 93117

Abstract

Design by Contract is a software engineering practice that allows semantic information to be added to a class or interface to precisely specify the conditions that are required for its correct operation. The basic constructs of Design by Contract are method preconditions and postconditions, and class invariants.

This paper presents a detailed design and implementation overview of jContractor, a freely available tool that allows programmers to write "contracts" as standard Java methods following an intuitive naming convention. Preconditions, postconditions, and invariants can be associated with, or inherited by, any class or interface. jContractor performs on-the-fly bytecode instrumentation to detect violation of the contract specification during a program's execution. jContractor's bytecode engineering technique allows it to specify and check contracts even when source code is not available. jContractor is a pure Java library providing a rich set of syntactic constructs for expressing contracts without extending the Java language or runtime environment. These constructs include support for predicate logic expressions, and referencing entry values of attributes and return values of methods. Fine grain control over the level of monitoring is possible at runtime. Since contract methods are allowed to use unconstrained Java expressions, in addition to runtime verification they can perform additional runtime monitoring, logging, and testing.

[1] Email: parkera@cs.ucsb.edu
[2] Email: muratk@ti.com

1 Introduction

Design by Contract (DBC) is the software engineering practice of adding semantic information to an application interface by specifying assertions about the program's runtime state. These assertions, collectively called a contract, must hold true at well-specified check-points during the program's execution. A method precondition is the portion of the contract which specifies the state that must be satisfied by the caller of the method. Invariants and method postconditions provide the other half of the contract, specifying the relevant state information that holds true upon completion of the method's execution.

A contract specifies the conditions that govern the correct usage and implementation of a module's interface. It is natural to express the contract as specification code that is a compiled along with the actual implementation code. The contract code can be evaluated to ensure that the module is operating according to specification, but correct execution of a program should not rely on the presence or checking of contract code. It is still desirable to automatically perform contract checking during a program's execution.

The idea of associating boolean expressions (assertions) with code as a means to argue the code's correctness can be traced back to Hoare [4] and others who worked in the field of program correctness. Meyer introduced Design by Contract as a built-in feature of the Eiffel language [9], allowing specification code to be associated with a class which can be compiled into runtime checks.

In this paper we present a detailed design and implementation overview of jContractor, distributed freely as a pure Java library which allows programmers to add contracts to Java programs as methods following an intuitive naming convention. A contract consists of precondition, postcondition, and invariant methods associated with any class or interface. Contract methods can be included directly within the Java class or written as a separate contract class. Contracts also work well with Java inheritance. Before loading each class, jContractor detects the presence of contract code patterns in the Java class bytecodes and performs on-the-fly bytecode instrumentation to enable checking of contracts during the program's execution.

Evaluating contracts as a program executes has some performance penalty. However, runtime contract checking is most useful during testing and debugging, when runtime speed is usually not critical. Contract code can remain in deployed bytecode, but contracts will only be evaluated when jContractor is invoked. Leaving contracts in deployed code helps with troubleshooting, but has no performance penalty when contracts are not checked.

The rest of the paper is organized as follows. In Section 2 we present a brief overview of jContractor from the perspective of a programmer using DBC in Java. In Section 3 we present details of jContractor's design and techniques for bytecode instrumentation to support runtime contract checking. Finally, we give a brief overview of related work, and discuss alternative uses

Table 1
Basic jContractor constructs

Construct	jContractor Pattern and Description
Precondition	protected boolean <methodname>_Precondition (<arguments>) Evaluated before a method is executed. Precondition failure indicates a bug in the caller.
Postcondition	protected boolean <methodname>_Postcondition (<arguments>, <return type> RESULT) Evaluated before returning from a method. Postcondition failure indicates a bug in the callee. The RESULT argument holds the method's return value.
Invariant	protected boolean _Invariant () Evaluated at the beginning and end of every public method. Invariant failure indicates a bug in the callee.
OLD	private <classname> OLD; return count == OLD.count + 1; Allows postconditions to refer to the state of an object at method entry.

of jContractor to perform other runtime monitoring.

2 jContractor overview

jContractor is available in Open Source form at `http://jcontractor.sourceforge.net`. It is a pure Java application, and will run on all platforms that support Java. In this section we provide an overview of how jContractor facilitates writing contracts and enables contract checking at runtime.

2.1 Writing contracts with jContractor

Contracts in jContractor are written as methods that follow a naming convention. The supported constructs and their patterns are described in Table 1. All contract methods return a boolean value, which is the result of the contract evaluation. If a contract method returns false, an exception will be thrown.

Preconditions

A precondition method takes the same arguments as the method it is associated with and is checked immediately before the method is executed.

It is the responsibility of the caller to ensure that the precondition check succeeds.

Postconditions

A postcondition method takes the same arguments as the method it is associated with followed by an additional RESULT argument of the same type as the method's return type. For void methods, RESULT is declared to be of type java.lang.Void. The postcondition associated with an instrumented method is checked immediately before the method returns with the RESULT argument holding the method's return value. This allows postconditions to make assertions about a method's result. It is responsibility of the class implementing the method to ensure that the postcondition holds.

Invariants

An invariant method is similar to a postcondition but does not take any arguments and is implicitly associated with all public methods. It is evaluated at the beginning and end of every public method. It is the responsibility of the implementation class that the invariant checks succeed.

Our approach differs slightly from Eiffel's invariant checking in that Eiffel invariants are only checked for method calls that originate outside of the class, as in foo.bar(). The invariant is not checked on the bar() method when it is called from another method in the same class. This distinction frees an Eiffel class from having to maintain the invariant during its internal operations.

OLD references

jContractor allows postconditions to refer to the state of an object at method entry. To enable this feature, a class must declare an instance variable named OLD of the same type as the class. Postconditions can then access attributes or execute methods by referencing OLD. Bytecode instrumentation routes all references to OLD to a clone of the object created at method entry. The clone is created using the clone() method, so the class must implement the java.lang.Cloneable interface and define this method.

Predicate Logic Quantifiers

jContractor also provides a support library for writing expressions using predicate logic quantifiers and operators such as Forall, Exists, suchThat, and implies. The supported quantifiers operate on instances of java.util.Collection, and are outlined in Table 2. These quantifiers offer a high level of abstraction and greatly improve readability when writing contract specifications.

Table 2
jContractor's logic constructs

Construct	jContractor Pattern and Description
ForAll	ForAll.in(collection).ensure(assertion) Ensures that all elements of a collection meet an assertion.
Exists	Exists.in(collection).suchThat(assertion) Ensures that at least one element of a collection meets an assertion.
Elements	Elements.in(collection).suchThat(assertion) Returns a java.util.Vector containing all the elements of a collection that meet an assertion.
Implies	Logical.implies(A, B) Evaluates true if A and B are both true, or if A is false. The logical equivalent of $\sim A \vee B$.

2.2 Checking contracts with jContractor

Running a program with contract checks enabled is as easy as passing the class name and command line arguments to the jContractor program. For example, a program containing jContractor contracts can be run with no runtime contract checking by

```
% java Foo arg1 arg2 arg3 ...
```

To run the same application with full contract checking one might enter

```
% java jContractor Foo arg1 arg2 arg3 ...
```

jContractor will replace the system class loader with a specialized class loader that will instrument class bytecodes as they are loaded, and execute the Foo program.

jContractor allows the user to specify the level of instrumentation (preconditions only, preconditions and postconditions, or all contracts) for each class in the system. To execute the Foo program checking only preconditions, but preconditions and postconditions in the bar package, and all contracts in class FooBar, one would enter

```
% java jContractor -pre * -post bar.* -all bar.FooBar Foo arg1
arg2 arg3 ...
```

jContractor supports wild cards similar to those used in Java import statements, allowing the user to concisely specify the instrumentation level. The

instrumentation levels may also be read from a file. The instrumentation levels (pre, post, and all) are those suggested by Meyer in [10, p. 393] and [9, p. 133]. The rationale is that for a method's postcondition to be satisfied, the precondition must have been satisfied. The invariant can only be satisfied if both preconditions and postconditions have been met. It is senseless to check the postcondition without checking the precondition, or the invariant without the precondition and the postcondition.

In some cases, it is not possible to replace the system class loader. For example, the class loader used by a web browser to load Java applets is beyond the programmer's control. In cases like these, jContractor can write instrumented bytecodes to disk, creating a set of instrumented classes that parallel the original uninstrumented classes. These instrumented classes can be executed without a full jContractor distribution. To instrument the file Foo.class one can execute jContractor's sister program, jInstrument

```
% java jInstrument Foo.class
```

This command will overwrite Foo.class with the instrumented bytecode.

3 Design and implementation

jContractor's basic operation involves discovering contracts associated with each class or interface just before the class bytecodes are loaded, and performing code transformations to enable contract checks at runtime.

jContractor instruments classes at the bytecode level, but the discussion in Sections 3.1 through 3.7 uses Java source code models to illustrate code transformations. Bytecode implementation details are discussed in Section 3.8.

In Section 3.1 we will discuss simple instrumentation techniques. In subsequent sections we will build upon the basic foundation to develop a robust Design by Contract implementation.

3.1 Implementing simple contract checks

The basic instrumentation technique used by jContractor is to execute the following steps on each class just before it is loaded. For each non-contract method m with signature s in class C

- Search for a method named m_Precondition with signature s in C or a separate contract class, C_CONTRACT, and prepend a call to m_Precondition to m.
- Search for a method named m_Postcondition with signature s, with an additional argument RESULT, in C or C_CONTRACT, and append a call to m_Postcondition to m.
- If m is public, search C and C_CONTRACT for a method named _Invariant.

```
java.util.Vector implementation;
...
public void push (Object o) {
  implementation.addElement(o);
}
```

Fig. 1. Listing of Stack.push(Object)

```
public void push (Object o) {
  if (!_Invariant())
    throw new InvariantViolationError();
  if (!push_Precondition(o))
    throw new PreconditionViolationError();

  implementation.addElement(o);

  if (!push_Postcondition(o, null))
    throw new PostconditionViolationError();
  if (!_Invariant())
    throw new InvariantViolationError();
}
```

Fig. 2. Simple instrumentation of Stack.push(Object)

Insert calls to the invariant method at the beginning and end of m.

Checking a contract at runtime involves calling the contract method and throwing an exception if the result is false. At first glance, checking a contract is quite straightforward. One need only add a call to the contract method at the beginning or end of the method, and throw an exception if the contact evaluates false. jContractor throws the following exceptions: PreconditionViolationError, PostconditionViolationError, and InvariantViolationError. All three extend java.lang.AssertionError.

To illustrate the instrumentation process, we use the push(Object) method of a Stack class, shown in Figure 1. A naive code transformation may result in the instrumented push(Object) method shown in Figure 2. This is almost correct, but overlooks an important point, which will be discussed in the next section.

3.2 The Assertion Evaluation Rule

Checking contracts at runtime usually helps find bugs and verify correctness. However, care must be taken to prevent contract checking itself from introducing bugs. Consider what happens when the invariant is checked in this simple example

```
class Stack {
  ...
  public int size () { ... }
  protected boolean _Invariant () {
    return size() >= 0;
  }
}
```

When the invariant is checked, the size() method is executed to ensure that the size is non-negative. size() is a public method, so the sample invariant should be checked at its entry point. But checking the invariant requires a call to size(), which leads to an infinite recursion of contract checks. To avoid situations like this, Design by Contract includes the Assertion Evaluation Rule [10, p. 402], which states that only one contract may be checked at a time. In the Stack example, the invariant will call size(). Since there is already a contract check in progress, the invariant will not be checked on size().

Implementing the Assertion Evaluation Rule requires that jContractor keeps track of when a contract check is in progress. This information is associated with each active thread. jContractor implements the Assertion Evaluation Rule by maintaining a shared hash table of threads that are actively checking contracts. Before a thread checks a contract, it queries the table to see if it is already checking one. If not, the thread inserts itself into the table, and proceeds with the contract check. When the check completes, the thread removes itself from the table.

The jContractorRuntime class provides static methods to determine if a thread is checking a contract, and to manage assertion checking locks on each thread. A Java model of the instrumented push(Object) method is shown in Figure 3.

It is necessary to wrap each contract check in a try-finally block so that the lock is released even if an exception is thrown while checking the contract. Usually, a contract violation terminates the program, in which case it doesn't matter if the lock is released or not. But the error could be caught and handled. If so, contract checking would halt for the remainder of the run if the lock were not released.

3.3 Predicate logic support

Contracts often involve constraints that are best expressed using predicate logic quantifiers. For example, in a graph structure there might be an array of nodes, each of which can have connections to other nodes. An implementation using this structure might want to ensure that each node in the graph is connected to at least one other. In mathematical notation, such a constraint could be written as

$$\forall n \in nodes \mid n.connections >= 1$$

```
public void push (Object o) {
  if (jContractorRuntime.canCheckAssertion()) {
    try {
      jContractorRuntime.lockAssertionCheck();
      if (!_Invariant())
        throw new InvariantViolationError();
      if (!push_Precondition(o))
        throw new PreconditionViolationError();
    } finally {
      jContractorRuntime.releaseAssertionCheck();
    }
  }

  implementation.addElement(o);

  if (jContractorRuntime.canCheckAssertion()) {
    try {
      jContractorRuntime.lockAssertionCheck();
      if (!_Invariant())
        throw new InvariantViolationError();
      if (!push_Postcondition(o, null))
        throw new PostconditionViolationError();
    } finally {
      jContractorRuntime.releaseAssertionCheck();
    }
  }
}
```

Fig. 3. Final instrumented version of Stack.push(Object)

jContractor allows a contract to be any function that evaluates to a boolean, so the graph constraint could be written using a loop, as shown below

```
java.util.Collection nodes;
...
java.util.Iterator i = nodes.iterator();
while (i.hasNext()) {
  if (((Node) i.next()).connections < 1)
    return false;
}
return true;
```

Using a loop works, but it requires the programmer to rewrite the quantifier's logic for each use. jContractor offers high level programming abstractions for common predicate logic expressions using a simple Java library. jContractor's quantifiers are summarized in Table 2, and can be applied to any instance of java.util.Collection, which includes all standard Java data structures. This

Table 3
Assertions provided with jContractor

Assertion	Description
InstanceOf	Asserts that objects are of a certain runtime type.
Equal	Asserts that objects are equal. The programmer specifies if the comparison should be by reference or by value.
InRange	Asserts that a number fall between minimum and maximum bounds.
Not	Used to negate another assertion, as in new Not(new Equal(Foo)).

library is completely isolated from the main jContractor code, and can be used independently.

The graph invariant can be expressed using jContractor's syntax as follows

```
java.util.Collection nodes;
...
Assertion connected = new Assertion () {
   public boolean eval (Object o) {
     return ((Node) o).connections >= 1;
   }
};
return ForAll.in(nodes).ensure(connected);
```

This version is the same length as the version using a loop, but it makes the contract more explicit. Some commonly used assertions are provided in the package, and are summarized in Table 3. For example, it is very simple to ensure that every element of a collection conforms to certain runtime type, as shown in the code snippet below

```
ForAll.in(elements).ensure(new InstanceOf(Integer.class));
```

This type of assertion is useful for controlling the type of objects that can be stored in a data structure.

Implementing the logic for a quantifier in Java is quite easy since all common data structures implement the java.util.Collection interface, and provide iterators. The biggest obstacle is finding a clean way of passing code into the iterator object. jContractor solves this problem by introducing an Assertion interface, which declares the standard interface method, eval(Object), to test the assertion. The implementation of the ForAll quantifier is shown in Figure 4. The Exists and Elements quantifiers are implemented in a similar way.

Finally, the static implies method of the Logical class allows programmers to write expressions of the form A implies B, where A and B are of type boolean.

```
public interface Assertion {
  public boolean eval (Object o);
}

public class ForAll {
  protected java.util.Collection theCollection;
  public ForAll (Collection c) {
    theCollection = c;
  }
  public static ForAll in (java.util.Collection c) {
    return new ForAll(c);
  }
  public boolean ensure (Assertion a) {
    java.util.Iterator i = theCollection.iterator();
    while (i.hasNext()) {
      if (!a.eval(i.next())) {
        return false;
      }
    }
    return true;
  }
}
```

Fig. 4. jContractor's implementation of the ForAll quantifier

Such an expression is the logical equivalent of $\sim A \vee B$. Using jContractor syntax, the expression would be written Logical.implies(A, B).

3.4 Implementing RESULT

Postconditions often make assertions about the function's result, which means that the postcondition method must have a way of referring to the function's return value. Implementing this feature in jContractor requires that the result be captured and passed as an extra argument to the postcondition method. If the method's return type is void, RESULT is declared to be of type java.lang.Void. The runtime value of a Void RESULT will always be null.

The mechanics of supporting RESULT are simple. jContractor adds a new local variable to each method with a postcondition for storing the result. Bytecode instrumentation replaces the return instructions with instructions to save the return value to the result variable. Finally, the instrumentation ensures that the computed result is passed to the postcondition during contract checking. For void methods, there is no result to save, so a null value is passed to the postcondition. A Java model of the instrumented size() method is shown in Figure 5 to illustrate this transformation.

```
public int size () {
  int $result = implementation.size();
  if (jContractorRuntime.canCheckAssertion()) {
    try {
      jContractorRuntime.lockAssertionCheck();
      if (!size_Postcondition($result))
        throw new PostconditionViolationError();
    } finally {
      releaseAssertionCheck();
    }
  }
  return $result;
}
```

Fig. 5. Instrumentation example to support RESULT

3.5 Implementing OLD

Postconditions often express how an object's state was changed by the method's execution. Therefore, the postcondition must be able to refer to the state of the object just before executing the method. Eiffel provides the old keyword for this purpose. jContractor mimics Eiffel's old syntax by introducing the OLD instance variable. The syntax for both implementations is shown in Figure 6.

In order to access old values in jContractor, the class must explicitly declare a private instance variable called OLD, of the same type as the class. The variable is private because it has meaning only in the class in which it is declared. Subclasses will declare their own OLD variables.

There are two alternative approaches to supporting old references. The first technique is to simply move the code from the old reference to the top of the method and save the result. An example of this approach is shown in Figure 7. This technique is used successfully in some other Java DBC tools that rely on the presence of source code [2,6,8,11]. However, it is not feasible when instrumenting bytecode. jContractor works with any valid Java bytecode, even those that have run the gauntlet of obfuscators and optimizers. The possibility of heavily obfuscated or optimized code makes it extremely difficult, if not impossible, to extract the code that made up the OLD reference.

The second approach, used by jContractor, is to create a clone of the object before executing the method body. When jContractor instruments a postcondition method, it redirects references to OLD to the cloned copy that holds the object's state at method entry. jContractor uses the clone() method to create the copy, so all classes that contain OLD references must implement the java.lang.Cloneable interface. Figure 8 illustrates this approach.

Unfortunately, simply storing the cloned state in the OLD instance variable is not sufficient, because the value needs to be saved at the entry point of every

Eiffel

```
class STACK[G]
  feature
    push (new_object: G) is
    deferred
    ensure
      size_increased: size = old size + 1
    end
end -- class STACK
```

jContractor

```
public abstract class Stack {
  private Stack OLD;
  public abstract void push (Object o);
  protected boolean push_Postcondition () {
    return size() == OLD.size() + 1;
  }
}
```

Fig. 6. Comparison of Eiffel and jContractor syntax for old

```
public void push (Object o) {
  int $old_size = size();
  // Check precondition and entry invariant
  // Method body
  // Check postcondition using $old_size
  //    in place of OLD.size()
  // Check exit invariant
}
```

Fig. 7. Implementing OLD by selectively saving data

method that uses OLD in its postcondition. In the worst case scenario, OLD needs to be saved for every method call. This leads jContractor to adopt a stack based variant of the solution. When execution enters a method that needs to save OLD, jContractor creates a clone of the object, and pushes it onto a stack. When the method exits, the object is popped from the stack, and used to check the postcondition. jContractor's version of push(Object) using this implementation is shown in Figure 9.

```
public void push (Object o) {
  OLD = (Stack) clone();
  // Check precondition and entry invariant
  // Method body
  // Check postcondition. OLD holds state at entry.
  // Check exit invariant
}
```

Fig. 8. Implementing OLD with a clone

```
public void push (Object o) {
  jContractorRuntime.pushState(clone());
  // Precondition and entry invariant check
  // Method body
  // Postcondition and exit invariant check
}
protected boolean push_Postcondition (Object o,
                                      Void RESULT) {
  Stack $old = (Stack) jContractorRuntime.popState();
  return count() == $old.size() + 1;
}
```

Fig. 9. Implementing OLD with a stack

3.6 Separate contract classes

jContractor allows contracts to be written in separate contract classes. Contract classes follow the naming convention classname_CONTRACT. When instrumenting a class, jContractor will find its contract class and copy all the contract code into the non-contract class. If the same contract is defined in both classes (both classes define a precondition for a method, for example), the two are logical and-ed together. Defining a contract in a separate class allows jContractor to add contracts to classes for which source code is not available.

3.7 Inheritance and contracts

jContractor's implementation of Design by Contract works well with both class and interface inheritance. A class inherits contracts from its superclass and implemented interfaces. A method's contract is made up of four parts, described in Table 4. Figure 10 shows how all the pieces are combined to form the complete contract. Any instrumented method must

- Accept all input and precondition states valid to the superclass method
- Meet all postcondition guarantees of the superclass method

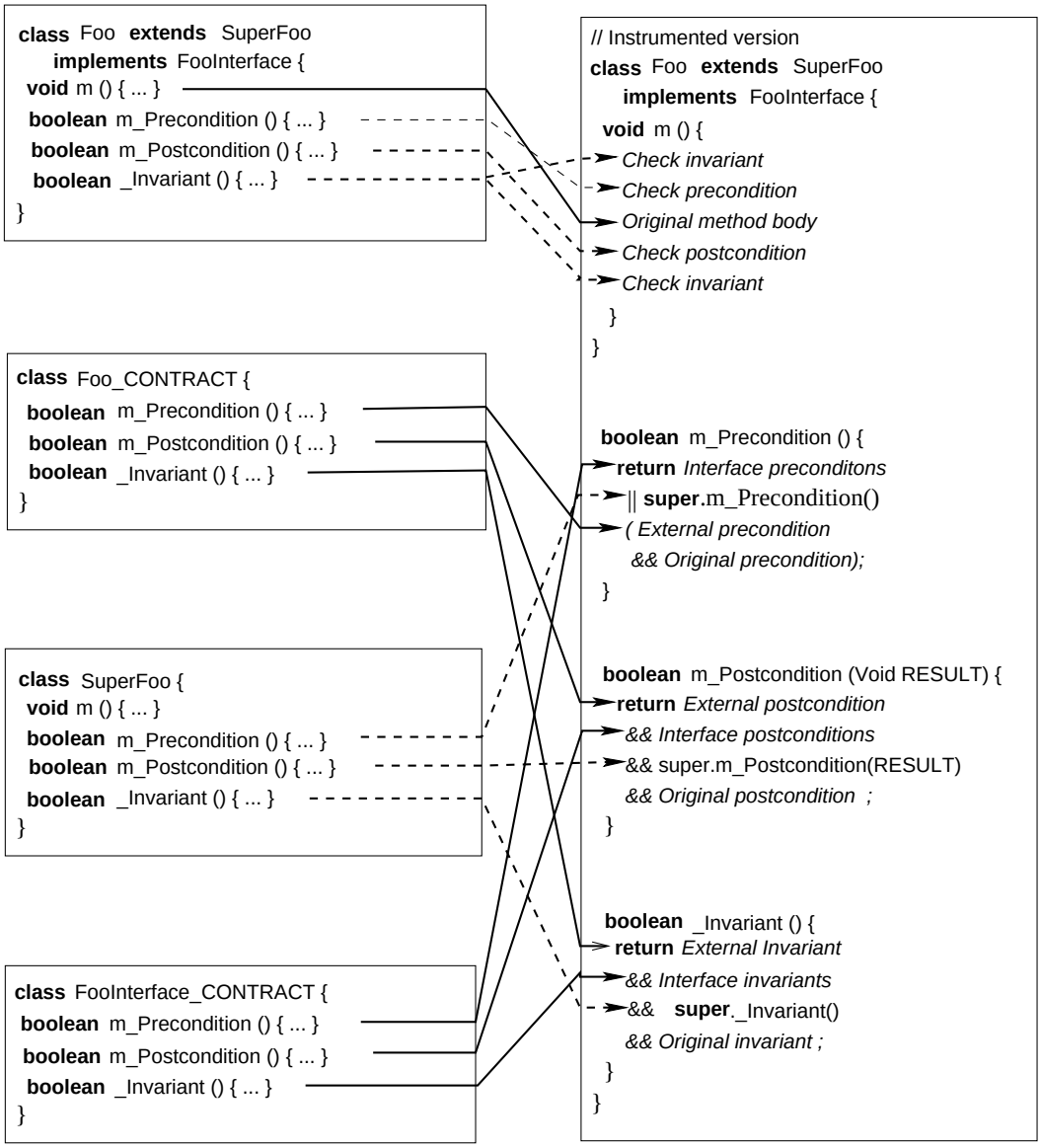

Fig. 10. How contracts are combined. Solid lines show where bytecode is copied, dotted lines show where method calls are inserted.

Put another way, the subclass method can only "weaken the precondition" and "strengthen the postcondition." This means that the precondition for the

Table 4
The four parts of a method's contract

Internal contract	Defined in the same class as the method.
External contract	Defined in a separate contract class.
Superclass contract	Inherited from the superclass.
Interface contracts	Inherited from interfaces.

```
class A {
  public int foo () { ... }
  protected boolean foo_Postcondition (int RESULT) { ... }
}
class B extends A {
  public int foo () { ... } // Overrides A.foo()
  protected boolean foo_Postcondition (int RESULT) {
    return super.foo_Postcondition(RESULT) && (RESULT > 0);
  }
}
```

Fig. 11. An example of postcondition inheritance

subclass method is logical or-ed with the super class precondition, and the postcondition is logical and-ed. This requires that the subclass accept all input valid to the superclass method, and may accept more that is invalid to the superclass. However, it must abide by the guarantees made by its parent, though it may strengthen those guarantees. Like postconditions, class invariants are logical and-ed. jContractor implements contract inheritance by instrumenting each contract method to call the superclass contract method. Figure 11 shows an example of this instrumentation.

Another approach to implementing contract inheritance is to copy the contract method from the superclass into the subclass. However, we feel that this approach is not as clean as calling the superclass method. More importantly, problems arise when contracts refer to private members in the superclass. Copying the contract code into the subclass would cause an illegal access error. Calling the superclass contract evaluates the contract in the context of the superclass, and handles private members correctly.

However, the technique described above does not work for interfaces, which cannot include contract code. Interface contracts must be written in separate contract classes, which jContractor will find and merge into the implementing class.

Inherited contracts guarantee that when a method is called on an object, the contracts will be met, regardless of the runtime type of the object. However, this guarantee is only meaningful for methods that behave polymorphically. Private methods, constructors, and static methods do not behave this way, so contract inheritance does not make sense for these methods. jContractor recognizes this, and only enforces inherited contracts for non-private, non-static, non-constructor methods.

3.8 Implementation of bytecode instrumentation

Our discussion of contract checking so far has illustrated code transformations using Java source code models. The actual instrumentation, however, is done using Java class bytecodes, and matches the logic of source code trans-

formation. jContractor uses the Byte Code Engineering Library (BCEL) [1] to instrument classes at the bytecode level, without requiring source code or additional compilation. Figure 12 shows the source code for pop() and its postcondition, and Figure 13 shows the disassembled bytecode for these methods (see [7] for an explaination of the instruction set). Instrumented versions of these methods are given in Figures 14 and 15. For brevity, this method has not been instrumented to check an invariant, and does not have a precondition. The patterns for checking preconditions and invariants are very similar.

Since a Java method can contain any number of return statements, jContractor replaces all return instructions with a jump to the end of the method, where code is inserted to check the postcondition and exit invariants.

jContractor uses the clone() method to allow postconditions to refer to the entry values of members. However, this creates a dependency between postconditions and clone(). Attempting to save an object's state while evaluating a postcondition called from clone() or from a method called by clone() would cause an infinite recursion. Since the clone() method is required to check contracts, contracts cannot be checked while executing clone() itself.

Statements 13–26 in Figure 14 make up the original pop() method. Note that the return instruction has been replaced with a jump to the end of the method. (This jump could be eliminated, but jContractor does not perform such optimization.) Statement 29 saves the result to the local variable $result, which jContractor adds to the method. Statements 30 and 33 check to see if the postcondition should be checked. If so, the the postcondition is checked by statements 39–44. Statements 47–59 are executed when the postcondition fails, and 60–64 are executed when the postcondition passes.

At this point, all of the major issues involved in runtime contract checking have been discussed. The ultimate goal (adding contract checking code to a class) is acomplished by way of small and largely independent subgoals (for example, adding code to check a precondition or handling old references). jContractor takes an assembly line approach to bytecode instrumentation. Each idependent operation is coded as a separate class, extending an abstract Transformation class. Then the class file to be instrumented is processed by each Transformation object, and at the end of the sequence it emerges fully instrumented. Figure 16 shows the sequence of transformations applied by jContractor.

This architecture is simple, but effective. Most of the transformations are completely independent. A few, however, need to save data for subsequent transformations. A shared hash table is created, into which a transformation can put data to be read later by another transformation. This solution is satisfactory for jContractor, but offers much opportunity for improvement. A generally useful framework would provide a more controlled mechanism to allow transformations to exchange data.

```
public Object pop () {
  return implementation.remove(size() - 1);
}
protected boolean pop_Postcondition (Object RESULT) {
  return (RESULT != null) && (size() == OLD.size() - 1);
}
```

Fig. 12. Listing of Stack.pop() and postcondition

```
public Object pop()
0:  aload_0
1:  getfield        Stack.implementation Ljava/util/Vector; (4)
4:  aload_0
5:  invokevirtual   Stack.size ()I
8:  iconst_1
9:  isub
10: invokevirtual   java.util.Vector.remove (I)Ljava/lang/Object;
13: areturn

Local variables:
index = 0 : Stack this

protected boolean pop_Postcondition(Object RESULT)
0:  aload_1
1:  ifnull          #24
4:  aload_0
5:  invokevirtual   Stack.size ()I
8:  aload_0
9:  getfield        Stack.OLD LStack;
12: invokevirtual   Stack.size ()I
15: iconst_1
16: isub
17: if_icmpne       #24
20: iconst_1
21: goto            #25
24: iconst_0
25: ireturn

Local variables:
index = 0 : Stack this
index = 1 : Object RESULT
```

Fig. 13. Bytecode listing of uninstrumented Stack.pop() and Stack.pop_Postcondition

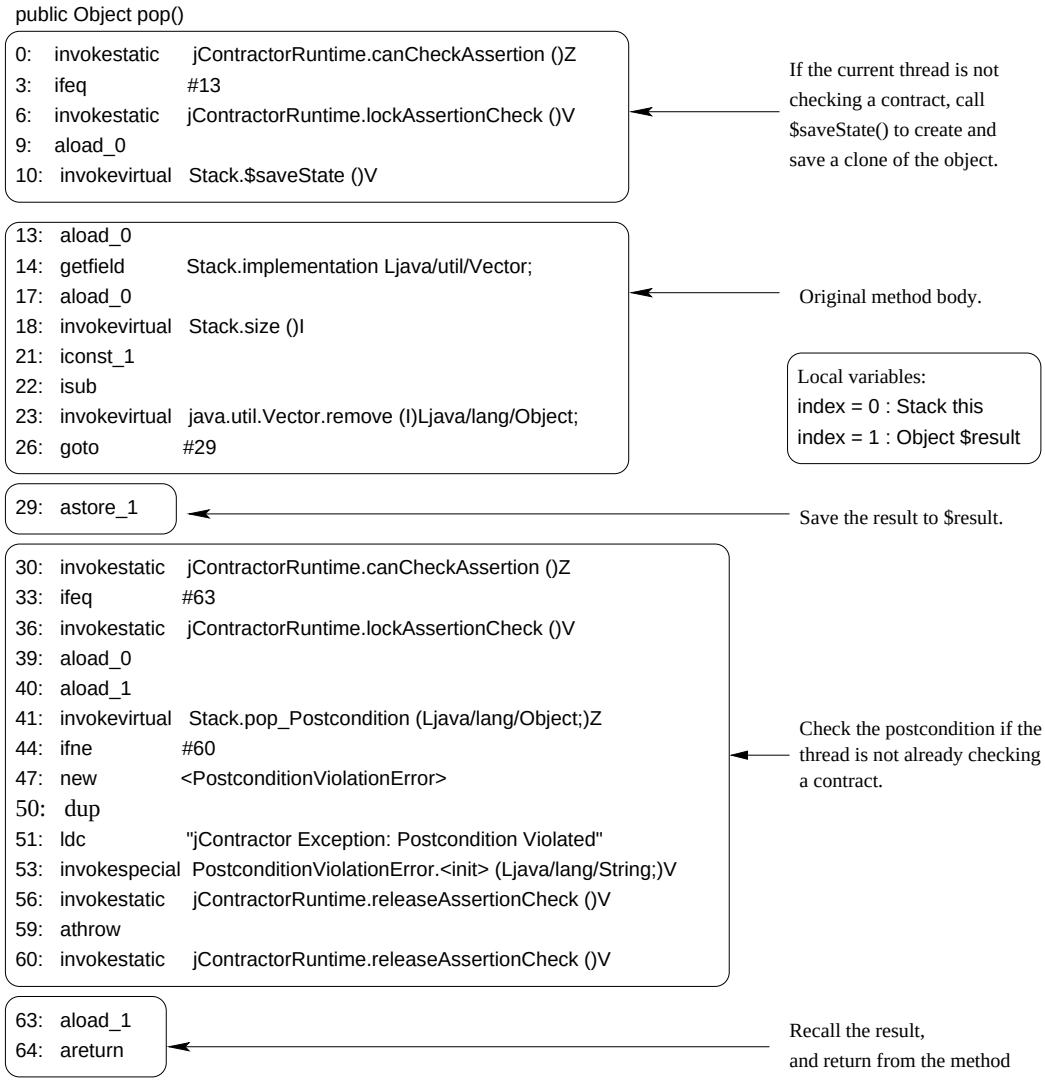

Fig. 14. Bytecode listing of instrumented Stack.pop()

4 Future and related work

4.1 Factory style instantiation

The implementation described in this paper allows the user to control instrumentation down to the class level. However, it is possible to control instrumentation on an instance-by-instance basis, using a factory model to instrument classes. Instead of creating an object with the new keyword, the client could invoke the jContractor.create(String, Class[], Object[]) method, which will instantiate and return an instrumented instance of the class.

Factory style instantiation can be implemented with slight modification to jContractor's current bytecode transformations. First, jContractor will create a new class that is a subclass of the base class. Then a method will be added to the subclass for each non-private method of the superclass with an associated contract. The bodies of these methods will simply wrap contract checking

```
protected boolean pop_Postcondition(Object RESULT)
0:   invokestatic     jContractorRuntime.popState ()Ljava/lang/Object;
3:   checkcast        <Stack>
6:   astore_2
7:   aload_1
8:   ifnull           #28
11:  aload_0
12:  invokevirtual    Stack.size ()I
15:  aload_2
16:  invokevirtual    Stack.size ()I
19:  iconst_1
20:  isub
21:  if_icmpne        #28
24:  iconst_1
25:  goto             #29
28:  iconst_0
29:  goto             #32
32:  ifeq             #39
35:  iconst_1
36:  goto             #40
39:  iconst_0
40:  goto             #43
43:  ifeq             #50
46:  iconst_1
47:  goto             #51
50:  iconst_0
51:  ireturn

Local variables:
index = 0 : Stack this
index = 1 : Object RESULT
index = 2 : Stack $old
```

Fig. 15. Bytecode listing of instrumented Stack.pop_Postcondition

code around a call to the superclass method. Finally, jContractor creates an instance of the instrumented class using the Java reflection API, and returns the object to the client. Thanks to polymorphism, the client can treat the instrumented object just as if it were the real thing, and all contracts will be checked. Figure 17 gives an example of this process.

This approach suffers from a few limitations. Private methods cannot be instrumented, because they are not visible in the subclass. Final classes can not be instrumented, because they cannot be subclassed. Also contracts from a separate contract class could refer to private members that are inaccessible to the subclass. These difficulties aside, a factory approach also requires

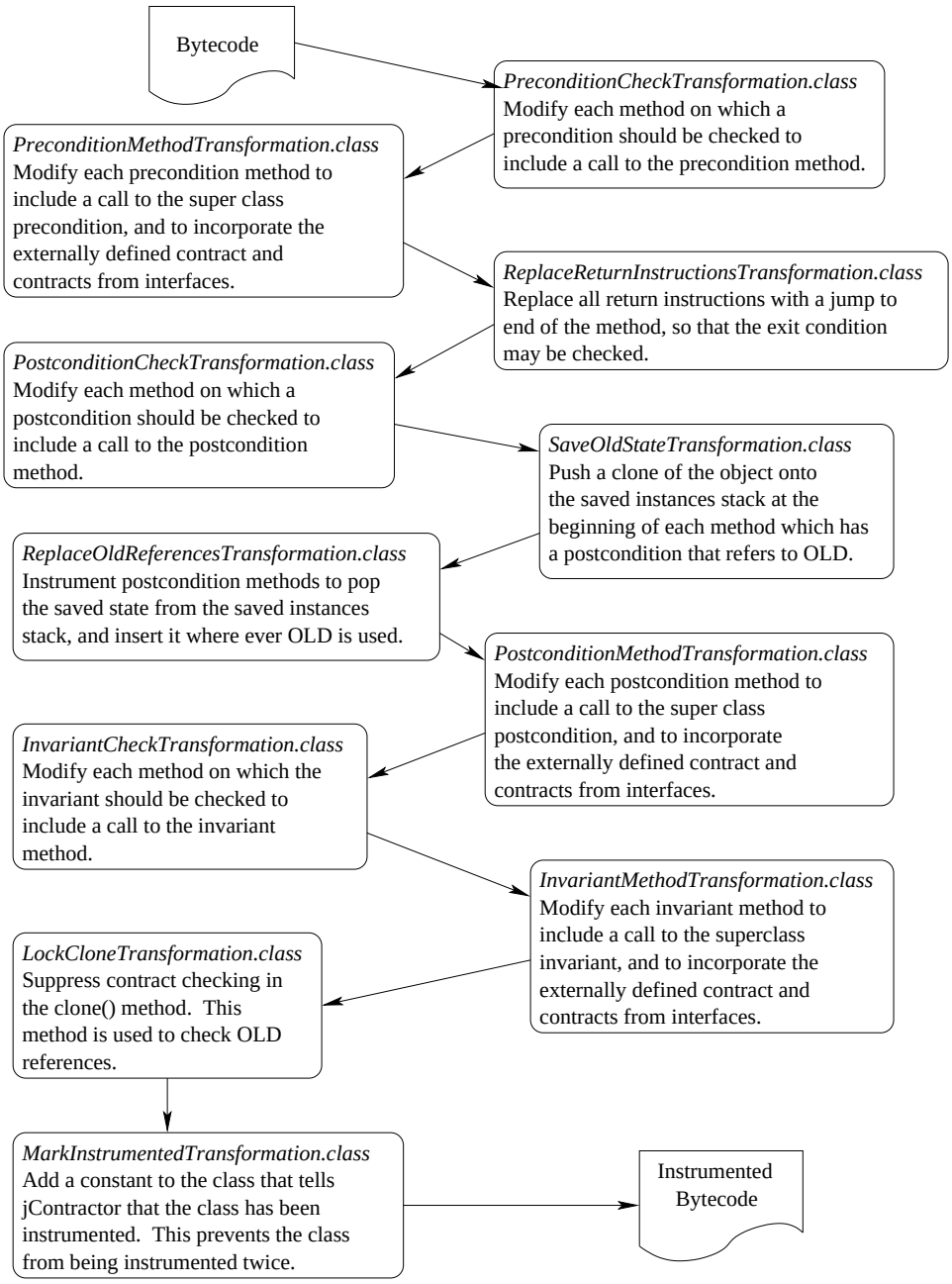

Fig. 16. Byte code transformation

the programmer to explicitly control instrumentation. However, the value of being able to create instrumented instances using a factory outweighs these drawbacks. Factory instantiation would give the programmer complete control over instrumentation, and could be useful for permanently enabling contracts in an isolated part of the code, or for programmatically controlling contract checks.

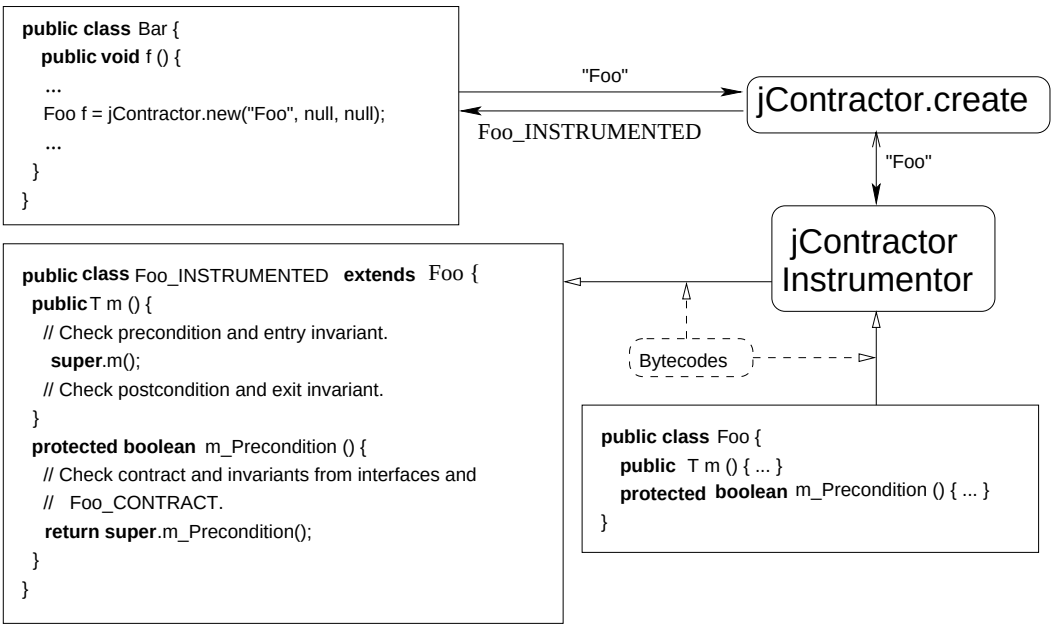

Fig. 17. Factory instrumentation and instantiation

4.2 Exception handling

The original jContractor proposal [5] outlined a mechanism to express contracts to handle method exceptions. However, the jContractor implementation does not yet support this feature.

A method's postcondition describes the contractual obligations of the method when it terminates successfully. When a method terminates abnormally due to some exception, it is not required to ensure that the postcondition holds. It is desirable, however, for the method to specify what conditions must still hold true in these situations, and to get a chance to restore the state to reflect this.

jContractor allows exception handlers to be associated with each method. An exception handler method's name is obtained by appending the suffix _OnException to the method's name. The exception handler method takes a single argument, of an exception type. The body of the method can include arbitrary Java statements and refer to the object's internal state using the same scope and access rules as the original method. When an exception is thrown in the original method, the exception handler will execute and attempt to correct the error or re-throw the exception.

4.3 Related work

Design by Contract originated in the Eiffel language, and has been implemented in many others. There are several tools available that support DBC for Java. However, most require source code availability, or use a special language to write contracts. jContractor allows programmers to write contracts

in pure Java, and can instrument classes even when the source code is not available.

Duncan and Hölzle describe Handshake [3], a dynamically linked library that intercepts JVM file accesses, and instruments classes on the fly. Handshake does not require source code for the classes that it instruments; the programmer specifies contracts in a separate file, using a Java-like syntax. This approach allows Handshake to add contracts to final classes, to interfaces, and to system classes. jContractor is unable to instrument system classes, due to restrictions in the system class loader.

Kramer's iContract [6] is a source code preprocessor that allows programmers to embed contracts in comments using the @pre, @post, and @invariant tags. This approach tightly couples specification and documentation, and allows contracts to be easily extracted by JavaDoc-style tools. iContract also supports the Forall and Exists quantifiers. iContract offers a clean and convienient syntax, but requires source code availability.

Jass [2] is another tool that supports Design by Contract using a source code preprocessor. In adition to preconditions, postconditions, and invariants, Jass supports loop variants and invariants, predicate logic quantifiers, and Eiffel-style "rescue-retry" exception handling. Jass provides a more robust mechanism to control how a class is used in an inheritance heirarchy than most other Design by Contract implementations, and a mechanism to specify the instance variables that a method is a allowed to change. Jass also supports "trace assertions," which express constraints on the order in which events occur.

JMSAssert [8], from Man Made Systems, also allows contracts to be embedded in comments. However, rather than acting as a preprocessor that outputs instrumented Java code, JMSAssert compiles embedded contracts into JMScript, a Java based scripting language. The contracts are checked using a DLL that extends the JVM. This approach leaves the orignal source code and bytecode unmodified. However, using a dynamically linked library creates a dependence on the operating system, and JMSAssert is currently only available for Microsoft Windows.

Finally, the Parasoft Corporation produces JContract [11], and a complementary unit testing and static analysis tool called JTest. Like iContract and Jass, JContract contracts are specified in comments. JContract includes the ForAll and Exists quantifiers. In addition to the standard DBC constructs, JContract allows the programmer to express contracts that control how a method is used in a multithreaded application, and provides a logging mechanism.

5 Conclusion

In this paper we describe the design and implementation of a pure Java library, jContractor, which requires no special tools such as modified compilers,

modified JVMs, or pre-processors to support Design by Contract. jContractor allows programmers to express contracts using pure Java in the form of precondition, postcondition, and invariant methods. Contract methods can be added to any Java class or interface or provided in a separately compiled contract class. jContractor introduces a novel bytecode engineering technique which allows it to check contracts even when the source code is not available.

Since contract methods are allowed to use unconstrained Java expressions, in addition to runtime contract checking they can perform additional runtime monitoring, verification, logging, and testing. For example, the code snippet below shows how jContractor could be used as a logging tool. jContractor also allows this code to be easily enabled and disabled by turning contract checking on and off. However, jContractor was designed to implement Design by Contract, and some of its features (support for inheritance, for example) may not be appropriate in other domains.

```
protected boolean push_Precondition (Object o) {
  System.out.println("Pushing " + o + "...");
  return true;
}
```

jContractor provides a rich set of syntactic constructs useful for expressing powerful contract specifications without extending the Java language or runtime environment. These include support for predicate logic expressions, the ability to refer to the state of the object at method entry (old), and the ability to refer to the computed result value for postcondition evaluation. A major advantage of jContractor's pure library based approach is that programmers are free to use their standard development tools and environments, and can also further extend jContractor's capabilities.

Allowing fine grain control over the level of monitoring at runtime adds great flexibility to the software development, testing and deployment cycles. Leaving the contract code within deployed class bytecodes results in no extra runtime performance penalties, but can assist greatly in field tests and troubleshooting.

jContractor has been released under the Apache Open Source License, and is available for download from `http://jcontractor.sourceforge.net`.

References

[1] Dahm, M., *Byte Code Engineering with the BCEL API*, Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin (1998).

[2] Detlef Bartetzko, M. M., Clemens Fischer and H. Wehrheim, *Jass - java with assertions*, , **55** (2001).

[3] Duncan, A. and U. Hölzle, *Adding Contracts to Java with Handshake*, Technical Report TRCS98-32, Department of Computer Science, University of California,

Santa Barbara (1998).

[4] Hoare, C., *An Axiomatic Basis for Computer Programming*, Communitications of the ACM **12** (1969).

[5] Karaorman, M., U. Hölzle and J. Bruno, *jContractor: A Reflective Java Library to Support Design By Contract*, in: *Proceedings of Meta-Level Architectures and Reflection, 2nd International Conference, Reflection '99. Saint-Malo, France. Lecture Notes in Computer Science #1616, Springer Verlag*, 1999, pp. 175–196.

[6] Kramer, R., *iContract - The Java Design by Contract Tool*, in: J. G. Madhu Singh, Bertrand Meyer and R. Mitchell, editors, *Proceedings of TOOLS USA '98, Santa Barbara, California, August 3-7, 1998*, 1998.

[7] Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, Reading, 1999.

[8] *Design by Contract for Java Using JMSAssert*, Technical report, Man Machine Systems.
URL `http://www.mmsindia.com/DBCForJava.html`

[9] Meyer, B., "Eiffel: The Language," Prentice Hall, New York, 1992.

[10] Meyer, B., "Object Oriented Software Construction, 2nd ed." Prentice Hall, Upper Saddle River, 1997.

[11] *Using Design by Contract to Automate Java Software and Component Testing*, Technical report, Parasoft Corporation.
URL
`http://www.parasoft.com/jsp/products/article.jsp?articleId=579&product=Jcontract`