Formal Methods in System Design, 27, 275–312, 2005

jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation

MURAT KARAORMAN muratk@ti.com
Texas Instruments, Inc., 315 Bollay Drive, Santa Barbara, CA 93117, USA

PARKER ABERCROMBIE parkera@cs.ucsb.edu
College of Creative Studies, University of California, Santa Barbara, CA 93106 USA

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 analysis.

Keywords: jContractor, Design by Contract, Java, bytecode instrumentation

1. Introduction

Design by Contract (DBC) is the software engineering practice of adding semantic information to an application interface by specifying monitored 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 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 [10] and others who worked in the field of program correctness. Meyer introduced Design by Contract as a built-in feature of the Eiffel language [15], allowing specification code to be associated with a class which can be compiled into runtime checks. In [16] Mitchell and McKim provide discussions on benefits as well as potential drawbacks of Design by Contract and introduce some design guidelines and principles for applying it to software engineering practice.

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 for any Java class or interface can be included directly within the 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.

Figure 1 depicts a typical user scenario where the system class loader has been replaced by the jContractor class loader to perform runtime contract monitoring. When engaged, before loading any class to the Java virtual machine the jContractor class loader transparently searches for the presence of jContractor DBC patterns in the compiled Java class bytecodes. Upon finding any contract associated with a method, the class loader instruments the class bytecodes to perform additional evaluation of the method's contract and to throw exceptions when a contract violation occurs during the method's invocation.

Evaluating contracts during program execution has some performance penalties. However, runtime contract checking is most useful during testing and debugging, when runtime

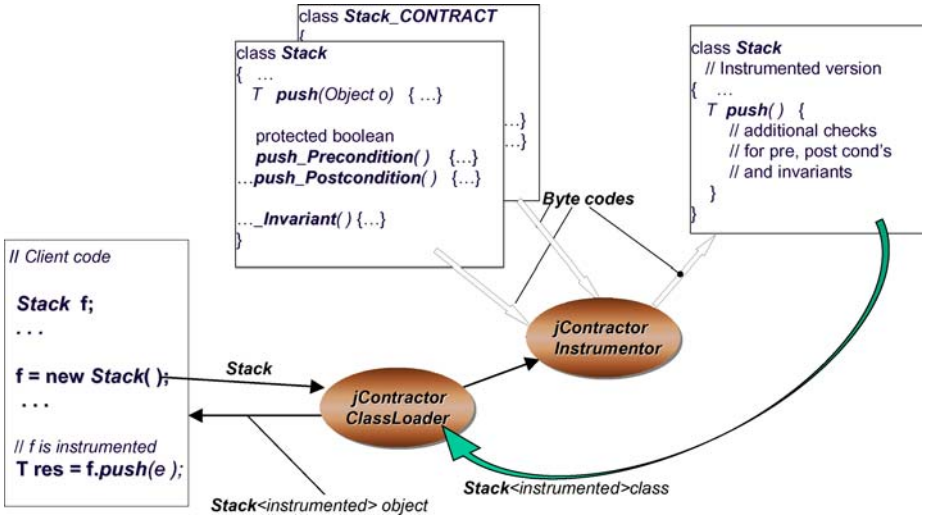

Figure 1. jContractor class loader performs on-the-fly bytecode instrumentation.

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 of jContractor to perform other runtime monitoring.

2. jContractor overview

jContractor is a 100% pure Java application library available as an open source project currently hosted 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 expressed as pure Java methods following a simple naming convention. jContractor provides runtime contract checking by instrumenting the bytecode of classes that define contracts. jContractor can either add contract checking code to class files to be executed later, or it can instrument classes at runtime as they are loaded. All contracts are written in standard Java, so there is no need to learn a special contract specification language. Contracts can be written in the class that they apply to, or in a separate contract class. This allows developers to add contracts to classes for which they do not have source code (3rd party libraries, for example). jContractor understands preconditions, postconditions, and class invariants, with full support for inheritance.

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. The discussion and examples in the section will refer to the Stack class shown in figure 2.

2.1.1. Preconditions. A precondition method takes the same arguments as the method it is associated with and returns a boolean. When contracts are monitored the precondition associated with a method is implicitly checked immediately before the method is executed. It is the responsibility of the caller to ensure that the precondition check succeeds.

Constructors are a small exception, since the call to the superclass constructor must be the first statement in a constructor. If such a call is present, it gets executed before the precondition is checked.

Preconditions for the Stack `push` and `searchStack` methods can be introduced by adding the following two methods to the Stack or Stack_CONTRACT class:

Table 1. Basic jContractor constructs.

CONSTRUCT	JCONTRACTOR PATTERN
Method Signature Standard Java class or interface method for which a jContractor design contract is written	`public` *returnType*$_i$ `aMethod(`*argList*$_i$`)`
Precondition Evaluated before a method is executed. Precondition failure indicates a bug in the caller	`protected boolean aMethod_Precondition` (*argList*$_i$)
Postcondition Evaluated before returning from a method. Postcondition failure. indicates a bug in the callee The `RESULT` argument holds the method's return value	`protected boolean aMethod_Postcondition` (*argList*$_i$ `,` *returnType*$_i$ `RESULT)`
Invariant Evaluated at the beginning and end of every public method. Invariant failure indicates a bug in the callee	`protected boolean _Invariant()`
Exception Handler *Currently not implemented.* Evaluated when the actual method terminates abruptly by throwing an exception. This method is then checked instead of any invariant or postconditions.	`protected` *returnType*$_i$ `aMethod_OnException` (*argList*$_i$, *Exception*$_i$ `e) throws` *Exception*$_i$
OLD Allows postconditions to refer to the state of an object at method entry	`private` *className* `OLD;` `return count == OLD.count + 1;`

```
protected boolean push_Precondition (Object o) {
  return o != null;
}
private boolean searchStack_Precondition (Object o) {
  return o != null;
}
```

Preconditions for constructors follow the same convention:

```
protected boolean Stack_Precondition (Object [] initialContents) {
   return (initialContents != null) &&
          (initialContents.length > 0);
}
```

```
class Stack implements Cloneable {
  private Stack OLD;
  private Vector implementation;

  public Stack () { ... }

  public Stack (Object [] initialContents) { ... }

  public void push (Object o) { ... }

  public Object pop () { ... }

  public Object peek () { ... }

  public void clear () { ... }

  public int size () { ... }

  public Object clone () { ... }

  private int searchStack (Object o) { ... }
}
```

Figure 2. Stack example.

Some additional rules about preconditions:

- Contract methods may not have preconditions.
- Native methods may not have preconditions.
- The `main(String [] args)` method may not have a precondition.
- The precondition for a static method must be static.
- The precondition for a non-static method must not be static.
- The precondition for a non-private method must be protected.
- The precondition for a private method must be private.

2.1.2. 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 back to the caller. jContractor's instrumentation logic implicitly assigns the actual result that is about to be returned by the method's execution to the `RESULT` argument. This allows postconditions to make assertions about a method's result. It is the responsibility of the class implementing the method to ensure that the postcondition holds.

An example postcondition method for the Stack `push` and `size` methods are shown below:

```
protected boolean push_Postcondition (Object o, Void RESULT){
    return implementation.contains(o) &&
           (size() == OLD.size() + 1);
}
protected boolean size_Postcondition (int RESULT) {
   return RESULT >= 0;
}
```

Postconditions for constructors follow the same convention. The return type for constructors is Void.

```
protected boolean Stack_Postcondition (Object [] initialContents,
                                       Void RESULT) {
   return size() == initialContents.length;
}
```

Postconditions may refer to the state of the object at method entry through the `OLD` instance variable as shown in the push postcondition. The actual mechanism is discussed in more detail below.

Finally, jContractor imposes the following rules about postconditions:

- Contract methods may not have postconditions.
- Native methods may not have postconditions.
- The postcondition for a static method must be static.
- The postcondition for a non-static method must not be static.
- The postcondition for a non-private method must be protected.
- The postcondition for a private method must be private.
- Postconditions for constructors cannot refer to OLD.

2.1.3. 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.

An example invariant for the `Stack` class:

```
protected boolean _Invariant () {
   return size() >= 0;
}
```

Following rules apply to jContractor invariants:

- Invariants are not checked for contract methods.
- Invariants are not checked for static methods.
- Invariants are not checked for native methods.
- Invariants are checked only at the exit of a constructor.
- The `_Invariant()` method must be declared protected and non-static.

2.1.4. 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. When execution enters the method, if there are any references to `OLD` in the method postcondition a clone of the object will be created and stored in `OLD`.

2.2. Exception handling

Exception handling support in Design By Contract is a controversial topic. The original jContractor proposal [11] outlined a practical mechanism to express contracts to handle method exceptions, however, the jContractor implementation does not yet support this feature. Plans and discussion to implement it are discussed in 4.2, in this section we outline the basic construct and the motivation.

A method's postcondition describes the contractual obligations of the method only when it terminates successfully. When a method terminates abnormally due to some exception, it is not required to ensure that the postcondition holds. It may still be 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, and controversially offer a hook to allow retry and rescue from the conditions that led to the exception.

jContractor defines a pattern for associating an exception handler with any class method. The exception handler method's name is obtained by appending the suffix `_OnException` to the method's name. The exception handler method has the same signature as the original method except for the addition of 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 and return a result or re-throw the exception.

If an exception handler is defined for a particular method, the exception handler must either re-throw the handled exception or compute and return a valid result. If the exception is re-thrown no further evaluation of the postconditions or class-invariants is carried out. If the handler is able to recover by generating a new result, the postcondition and class-invariant checks are performed before the result is returned, as if the method had terminated successfully.

2.3. Contract violations

When a contract is violated, an error is thrown, usually ending in program termination with an informative error message. The error classes are:

`edu.ucsb.ccs.jcontractor.PreconditionViolationError`,
`edu.ucsb.ccs.jcontractor.PostconditionViolationError`, and
`edu.ucsb.ccs.jcontractor.InvariantViolationError`.

While nothing prevents the caller of the method from catching and handling these errors, this practice is not recommended. A contract violation usually points to a bug that should fixed before release, not handled at runtime.

2.4. 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. Figure 3 shows a separate contract class written for the Stack example.

The separate contract class methods can reference the variables and methods of the class with which it is associated. However, to get the compiler to accept the code, it is sometimes necessary to provide fake variables and methods, such as `implementation` and `searchStack(Object)` in the contract class in figure 3. When jContractor instruments a class, the code for contracts defined in the separate contract class is inserted into the non-contract class. Once there, it will have access to any members and variables of the non-contract class. A convenient way to get access to non-private fields and methods in the non-contract class is to make the contract class a subclass of the non-contract class.

2.5. Contracts and inheritance

jContractor's implementation of Design by Contract works well with both class and interface inheritance. Contracts are inherited, just like methods. When a method is overridden in a subclass, that class may specify its own contracts to modify those on the superclass method.

```
class Stack_CONTRACT extends Stack {
  private Stack OLD;

  private Vector implementation; // Dummy variable

  protected boolean
  Stack_Postcondition (Object[] initialContents, Void RESULT) {
     return size() == initialContents.length;
  }

  protected boolean
  Stack_Precondition (Object [] initialContents) {
     return (initialContents != null) &&
            (initialContents.length > 0);
  }

  protected boolean push_Precondition (Object o) {
     return o != null;
  }
  protected boolean push_Postcondition (Object o, Void RESULT){
     return implementation.contains(o) &&
     (size() == OLD.size() + 1);
  }

  private int searchStack (Object o) { // Dummy method
     return 0;
  }

  private boolean searchStack_Precondition (Object o) {
     return o != null;
  }

  protected boolean _Invariant () {
     return size() >= 0;
  }
}
```

Figure 3. Separate contract class for stack.

jContractor instruments each method to enforce contract checking based on the following operational view.

A subclass method's contract must:

– Allow all input valid for its superclass method.
– Ensure all guarantees of the superclass methods.

Put another way, the method may "weaken the precondition and strengthen the postcondition". What this means is that during run-time evaluation of contracts the preconditions

for the subclass method are logical or-ed with the superclass preconditions, and the post-conditions are logical and-ed. This ensures that the subclass accepts all input valid to the super class method, and may accept more that is invalid to the superclass. However, it must abide by the guarantees made by its parent. Like postconditions, class invariants are logical and-ed. Additional design and implementation details can be found in Section 3.6.

Interfaces may also have contracts (provided in separate contract classes), so contracts are subject to multiple inheritance. Contracts from interfaces are logical or-ed with the superclass and subclass contracts in the case of preconditions. For post-conditions and invariants they are logical and-ed.

2.6. Predicate logic support

Contracts often involve constraints that are best expressed using predicate logic quantifiers. jContractor 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.

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

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

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

Table 2. jContractor's logic constructs.

CONSTRUCT	PATTERN and USAGE

ForAll
Check if all elements of a collection meet an assertion.
	`Collection aCollection;`
	`Assertion anAssertion;`
	`ForAll.in(aCollection).ensure(anAssertion)`

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

Assertion
Standard interface for designing assertions, by implementing eval method which takes an object param to evaluate.
	`public interface Assertion {`
	`  public boolean eval (Object o);`
	`}`

Operator
Standard interface for an object that transforms other objects. Using the ForAll quantifier, Operators allow easy application of some function to all the elements in a Collection.
	`public interface Operator {`
	`  public void execute (Object o);`
	`}`

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

Table 3. Standard jContractor assertions.

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))`.

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.

2.6.1. Using operators. The `Assertion` interface describes a test that evaluates to a `boolean`, but evaluating the `Assertion` should not modify the elements of the collection. We define the standard `Operator` interface to design operators to apply to and transform other objects. An `Operator` can be used with the `ForAll` quantifier for improved syntax as in the following example which defines and uses a new 'initialize' operator.

```
java.util.Collection elements;
// define new Operator
Operator initialize = new Operator () {
  void execute (Object o) {
    ((Node) o).init();
  }
};
...
// apply initialize Operator to all elements
ForAll.in(elements).execute(initialize);
```

2.7. Checking contracts with jContractor

Two utility applications are provided to instrument and run Java applications with runtime contract checking: `jContractor` and `jInstrument` (see Table 4.) The `jContractor` utility replaces the standard Java class loader and performs on-the-fly bytecode instrumentation to enable contract checks at runtime. The `jInstrument` utility on the other hand can be

Table 4. Standard jContractor utilities.

Usage	`jContractor` *[options] classname [cmdLineArgs]* `jInstrument` *[options] classname*
Options:	
`-d directory`	Specifies the directory in which instrumented class files should be saved (jInstrument only).
`-f file`	Specifies a file that holds instrumentation levels.
`--none <class or package>`	Suppresses instrumentation of the given class(es).
`--pre <class or package>`	Instruments the given class(es) for preconditions checks.
`--post <class or package>`	Instruments the given class(es) for precondition and postcondition checks.
`--all <class or package>`	Instruments the given class(es) for all contract checks.
`--verbose`	Prints the names of classes as they are instrumented.
`--version`	Prints the program version and exits.
`--help`	Prints this table and exits.

used to perform the same bytecode instrumentation to individual Java class files without running the Java application.

2.7.1. Using jContractor utility. The easiest way to run a Java program with runtime contract checking is to use the jContractor application and pass the class name with its 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. Any command line arguments that appear after the class name are passed to the `main(String [] args)` method of that class.

2.7.2. Using jInstrument utility. 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, the `jInstrument` utility makes it possible to add contract checking code to class files so that they may be run with any Java runtime environment. The instrumented classes then can be executed without a full jContractor distribution.

`jInstrument` takes the name of the class file as an argument and writes the instrumented file to the directory specified by the "-d" option, or the current directory by default, creating a set of instrumented classes that parallel the original uninstrumented classes. For example, issuing the following command:

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

overwrites the Java class file, `Foo.class`, with a jContractor instrumented version. Running the modified `Foo.class` results in execution with runtime contract checks:

```
% java Foo
```

2.7.3. Controlling the instrumentation level. jContractor allows the user to specify the level of instrumentation (preconditions only, preconditions and postconditions, or all contracts) for each class in the system. For example, 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 [17, p. 393] and [15, p. 133]. The supported instrumentation levels are:

– none - No contracts are checked.
– pre - Preconditions are checked.
– post - Preconditions and postconditions are checked.
– all - Preconditions, postconditions, and invariants.

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.

3. Design and implementation of jContractor

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 3.6 uses Java source code models to illustrate code transformations. Bytecode implementation details are discussed in Section 3.9.

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`, modify the method to have a single exit point and append a call to `m_Postcondition` to `m`.
- If `m` is public, search `C` and `C_CONTRACT` for a method named `_Invariant`. 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`.

To illustrate the instrumentation process, we use the `push(Object)` method of a `Stack` class, shown in figure 4. A naive code transformation may result in the instrumented `push(Object)` method shown in figure 5. This is almost correct, but overlooks an important point, which will be discussed in the next section.

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

Figure 4. 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();
}
```

Figure 5. Simple instrumentation of Stack.push (Object).

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 [17, 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 6.

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. Implementing predicate logic support library

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

```
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();
       }
   }
}
```

Figure 6. Final instrumented version of Stack.push (Object).

is shown in figure 7. 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. 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,

```
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;
  }
}
```

Figure 7. jContractor's implementation of the ForAll quantifier.

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 8 to illustrate this transformation.

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 9.

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 10. This technique is used successfully in some other Java DBC tools that rely on the presence of source code [7, 12, 14, 25].

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

Figure 8. Instrumentation example to support RESULT.

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;
   }
}
```

Figure 9. 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>
}
```

Figure 10. Implementing OLD by selectively saving data.

```
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>
}
```

Figure 11. Implementing OLD with a clone.

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 11 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 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 12.

Table 5. 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

```
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;
}
```

Figure 12. Implementing OLD with a stack.

3.6. Implementing support for contract inheritance

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 5. Figure 13 shows how all the pieces are combined to form the complete contract.

To ensure that the subclass method can only "weaken the precondition" and "strengthen the postcondition." the precondition for the subclass method is logical or-ed with the super class precondition, and the postcondition is logical and-ed. Like postconditions, class invariants are logical and-ed. jContractor implements contract inheritance by instrumenting each contract method to call the superclass contract method. Figure 14 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

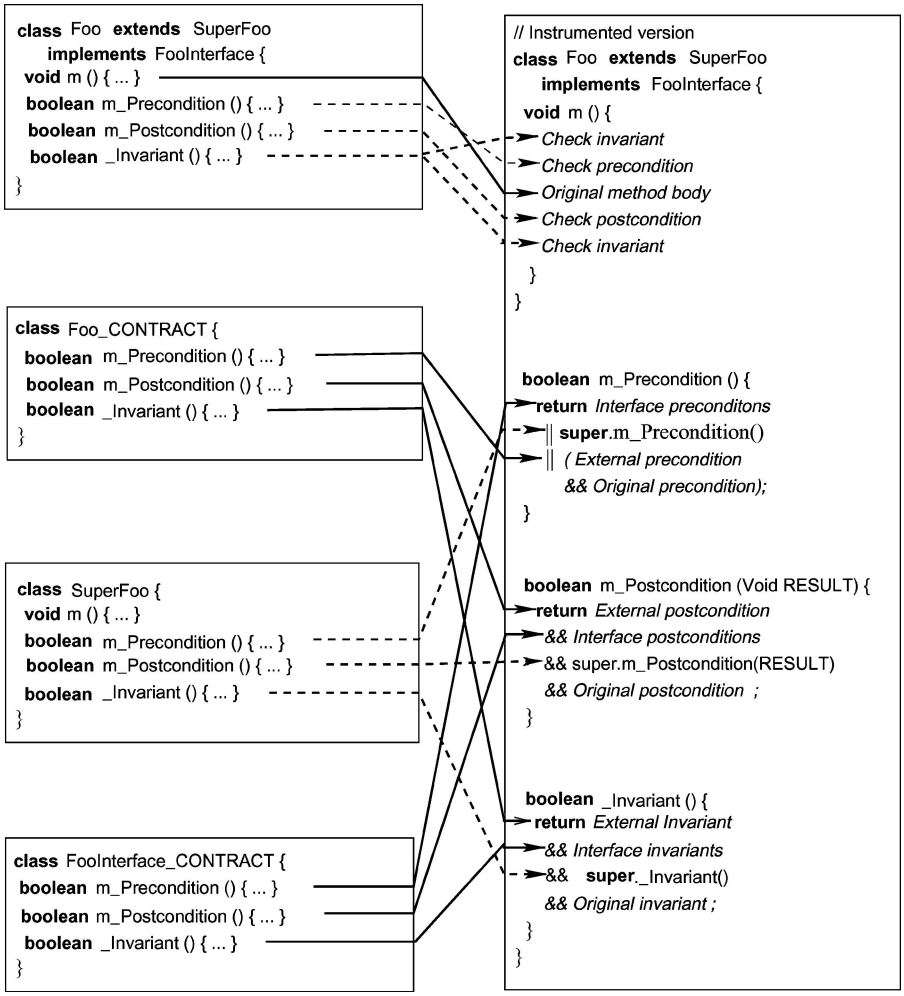

Figure 13. How contracts are combined. Solid lines show where bytecode is copied, dotted lines show where method calls are inserted.

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.

3.7. Supporting polymorphism

Contracts are essentially specifications checked at run-time. They are not part of the functional implementation code, and a "correct" program's execution should not depend on the

```
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);
  }
}
```

Figure 14. An example of postcondition inheritance.

presence or enabling of the contract methods. In the rest of this section we discuss the contravariance and covariance issues arising from the way contracts are inherited.

The inheritance of preconditions from a parent class follows contravariance: as a subclass provides a more specialized implementation, it should weaken, not strengthen, the preconditions of its methods. Any method that is redefined in the subclass should be able to at least handle the cases that were being handled by the parent, and in addition handle some other cases due to its specialization. Otherwise, polymorphic substitution would no longer be possible. A client of X is bound by the contractual obligations of meeting the precondition specifications of X. If during runtime an object of a more specialized instance, say of class Y (a subclass of X) is passed, the client's code should not be expected to satisfy any stricter preconditions than it already satisfies for X, irrespective of the runtime type of the object. jContractor supports contravariance by evaluating the a logical-OR of the precondition expression specified in the subclass with the preconditions inherited from its parents. For example, consider the following client code snippet:

```
X x; // class Y extends class X
Y y = new Y(); // Y object instantiated

x = y; // x is polymorphically attached to a Y object
int i = 5;

x.foo(i); // only Precondition(X,[foo,int i]) need by met
```

When executing x.foo(), due to dynamic binding in Java, class Y's the foo() method gets called, since the dynamic type of the instance is Y. If jContractor is enabled this results in the evaluation of the following precondition expression:

$$Precondition(X, [foo, int\ i]) \vee Precondition(Y, [foo, int\ i])$$

This ensures that no matter how strict *Precondition*(*Y,foo*) might be, as long as the *Precondition*(*X,foo*) holds true, `x.foo()` will not raise a precondition exception.

The inheritance of postconditions is similar: as a subclass provides a more specialized implementation, it should strengthen, not weaken the postconditions of its interface methods. Any method that is redefined in the subclass should be able to guarantee at least as much as its parent's implementation, and then perhaps some more, due to its specialization. jContractor evaluates the logical-AND of the postcondition expression found in the subclass with the ones inherited from its parents.

3.8. Contract specification anomalies

jContractor prevents a subclass from strengthening an overloaded method's precondition by logical-OR'ing any inherited preconditions with that of the subclass. This approach prevents program contract specification errors from violating contravariance. For example, consider the following precondition specifications for the `foo()` method defined both in X and Y, referring to the example code snippet in Section 3.7:

$$Precondition(X, [foo, int\ a]) : a > 0 \quad \text{(I)}$$

$$Precondition(Y, [foo, int\ a]) : a > 10 \quad \text{(II)}$$

From a specification point of view (II) is stricter than (I), since for values of $a : 0 < a <= 10$, (II) will fail, while (I) will succeed, and for all other values of a, (I) and (II) will return identical results. While perfectly accepted by jContractor, (I) and (II) illustrates an error in the program's contract specification. Subclass precondition, (II), is stronger than the parent's, (I). The call:

```
x.foo(5);
```

does not raise an exception using jContractor since it meets *Precondition(X,foo,int a)*. However, the implementor of class Y may perceive this as a problem since, due to Java's late binding, Y's method foo(int a) will get called with no precondition violations even though Y's foo precondition seems to be violated. Clearly the problem lies with the design of X and Y and their contract specifications. Theoretically these types of errors can be discovered using formal verification, by proving that the following logical-implication holds for each redefined method m():

$$Precondition(ParentClass, m) \rightarrow Precondition(SubClass, m)$$

For the previous example, it is easy to prove that (I) does not logically-imply (II). However, it is beyond the scope of jContractor to do formal verification for logical inference of specification anomalies.

A different type of behavior anomaly is also present due to the logical-OR'ing with parent's preconditions. Consider the same example code snippet in Section 3.7, but this time with a correct contract specification for the parent X and subclass Y:

$$Precondition(X, [foo, int\ a]) : a > 10 \quad \text{(I)}$$

$$Precondition(Y, [foo, int\ a]) : a > 0 \quad \text{(II)}$$

In this example, subclass Y follows contravariance correctly and weakens parent X's foo precondition, allowing the additional range values for a, $a : 0 < a <= 10$. However, the same call:

```
x.foo(5);
```

again does not raise an exception using jContractor. As with the earlier scenario, the subclass Y's precondition (II) gets evaluated since the reference x is bound to an object of runtime type Y. Since (II) is not violated the precondition evaluates to true. This time, however, the problem actually lies in the client's code. Declared type of x is X, so the client should respect the contract for class X. The client, however, calls x's foo method passing a value less than 10, violating X's contract, (I). If x had been bound at runtime to an object with the runtime type X jContractor would have caught the client's violation of the contract, however since the actual runtime binding is to an object whose runtime type is of a more specialized subclass, Y, whose precondition is met, jContractor allows the call without reporting any DBC exceptions. This can be considered a limitation of jContractor.

Similar specification anomalies could also occur when a subclass strengthens its parent's invariants, or weakens one of its postconditions. jContractor, can be extended in the future to detect and log diagnostic messages for design anomalies, say when any one of the logical-OR'ed precondition expressions evaluates to false. In the first scenario above, jContractor could log a diagnostic message that the precondition has been potentially illegally strengthened in the subclass, thus forcing the programmer to correct the precondition. In the second scenario, the logged message could be used to highlight a client side contract violation, which was safe for the particular runtime usage but potentially can lead to problems if a parent instance were substituted.

3.9. 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 transformation. jContractor uses the Byte Code Engineering Library (BCEL) [6] to instrument classes at the bytecode level, without requiring source code or additional compilation. Figure 15 shows the source code for `pop()` and its postcondition, and figure 16 shows the disassembled bytecode for these methods (see [13] for an explanation of the instruction set). Instrumented versions of these methods are given in figures 17 and 18. 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()`.

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

Figure 15. 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
```

Figure 16. Bytecode listing of uninstrumented Stack.pop() and Stack.pop_Postcondition.

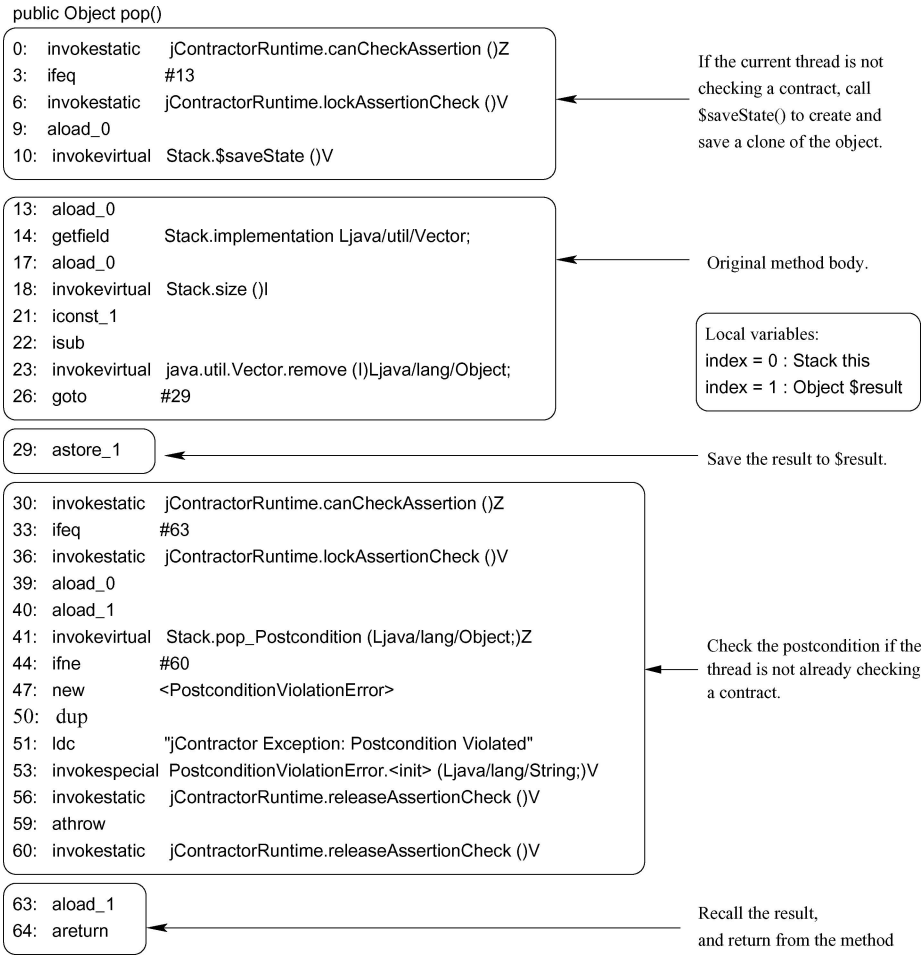

Figure 17. Bytecode listing of instrumented Stack.pop().

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. To illustrate this recursion, we present the following example:

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

```
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
```

Figure 18. Bytecode listing of instrumented Stack.pop_Postcondition.

```
public Object clone () {
  Stack other = new Stack();
  for (int i = 0; i < implementation.size(); i++) {
    if (implementation.elementAt(i) instanceof Cloneable){
      other.push(((Cloneable)implementation.
        elementAt(i)).clone());
    } else {
      other.push(implementation.elementAt(i));
    }
  }
  return other;
}
```

Suppose that a stack, `S1`, attempts to clone itself. The clone, `S2`, is created and an object is pushed onto it. Evaluating the postcondition of `push()` requires a reference to `OLD`, which

requires a call to `clone()`. `S2` attempts to clone itself, which creates `S3`. When an object is pushed onto `S3`, the postcondition will need to be evaluated, which requires a clone of `S3`. And so on. To avoid cases like this, jContractor does not check contracts while executing `clone()` methods. Contract checks are suppressed using the same mechanism used for Assertion Evaluation Rule. See Section 3.2 for a discussion of this mechanism.

Statements 13–26 in figure 17 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 19 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.

4. Future 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 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 20 gives an example of this process.

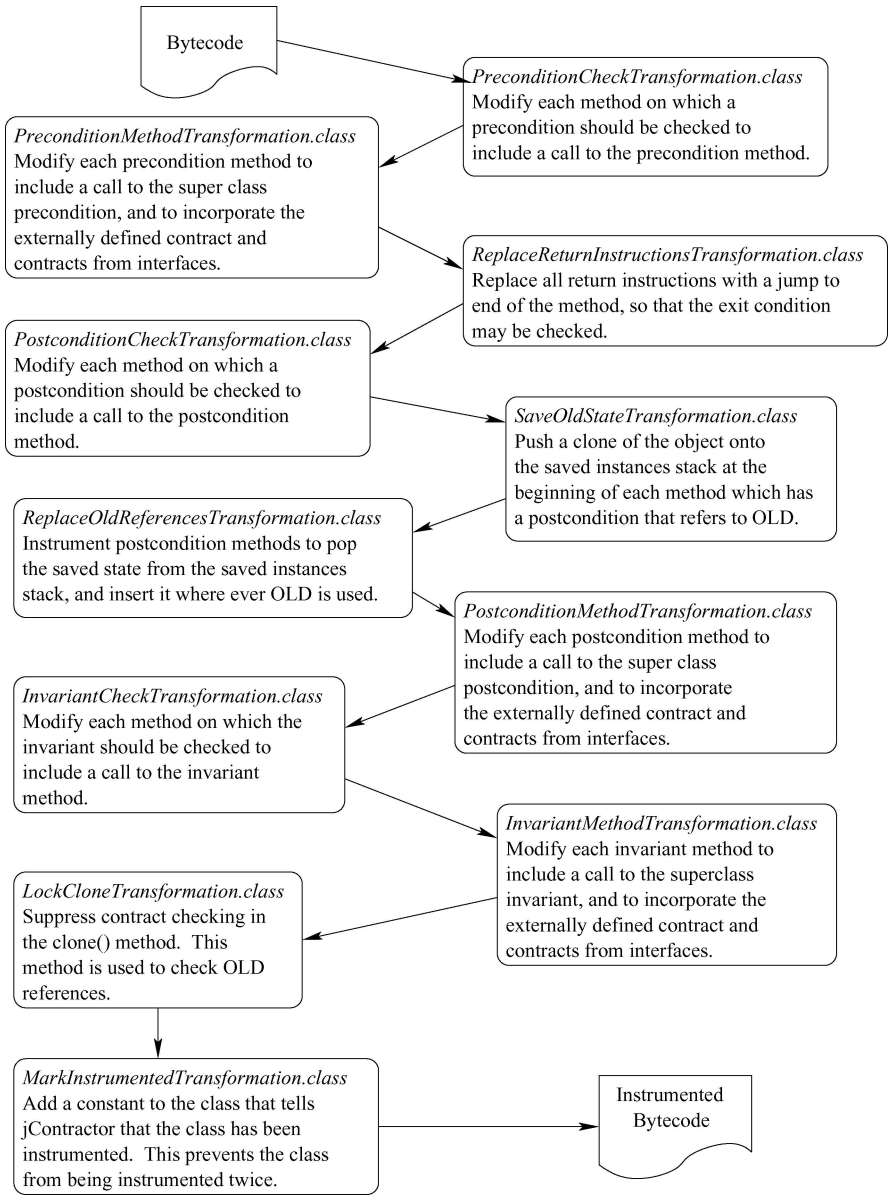

Figure 19. Byte code transformation.

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

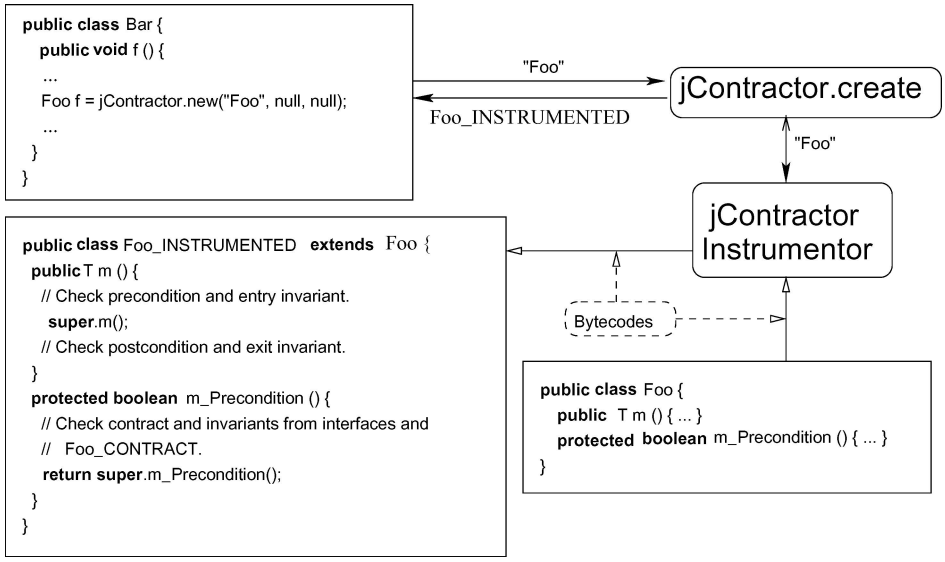

Figure 20. Factory instrumentation and instantiation.

also requires 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.

4.2. Exception handling

Meyer first introduced the rescue and retry mechanism to Eiffel Language as part of its built-in Design by Contract features to perform exception handling and recovery which only takes place when contract monitoring is enabled.

We proposed a jContractor pattern to support exception handling as part of an method's contract specification, which is outlined in Table 1 and explained in Section 2.2. We have left it largely as an unsupported feature due to the controversial nature of the feature. If such recovery from an exception condition is possible, it is better to incorporate this handler into the implementation of the method itself, which forestalls throwing the exception at all. The general support for exceptions as a language feature have been largely attributed to Goodenough [9]. Black [2] on the other hand provided a case against exception handling. We also refer the reader to [24] for an interesting presentation of some of the issues and capturing some of the debate on the topic.

The pattern is kept in the jContractor design, however, to illustrate that support can be added in the future and that this is not an inherently unsupportable Design by Contract mechanism in the design of jContractor.

jContractor can implement `OnException` handlers by instrumenting the method to add a wrapper around the code to catch exceptions thrown inside the original method body. If the contracts include an exception-handler method for the type of exception caught by the wrapper, the exception handler code gets executed.

4.3. State modification support: Change lists

Many programming languages distinguish between functions, which perform a computation and return a result, and procedures which modify the state of an object or of global variables. Often a method's contract will specify which fields of an object the method can modify. The contract will fail if any other fields are modified. This feature is implemented in the Jass tool [7].

This feature could be implemented by associating with each method a list of fields that the method is allowed to change. The method may only write to fields in it's "change list". If the list is empty, it may not write to any fields.

The first challenge to implementing this feature is devising a convenient syntax. One solution is to introduce a class of static "dummy" methods:

```
class ChangeList {
  public static void add (Object i) { }
  public static void add (int i) { }
  public static void add (short i) { }
  public static void add (float i) { }
  public static void add (double i) { }
  public static void add (byte i) { }
  public static void add (char i) { }
  public static void add (boolean i) { }

  public static void makeEmpty () { }
}
```

This class could be used in contracts as follows:

```
protected boolean push_Postcondition (Object o, Void RESULT) {
  ChangeList.add(implementation);
  ChangeList.add(size);
  ...
}

protected boolean size_Postcondition (int RESULT) {
  ChangeList.makeEmpty();
  ...
}
```

The contracts compile into the following bytecode:

```
protected boolean push_Postcondition(Object arg1, Void arg2)
0: aload_0
1: getfield ChangeListTest.implementation Ljava/util/Vector;
4: invokestatic ChangeList.add (Ljava/lang/Object;)V
7: aload_0
8: getfield ChangeListTest.size I
11: invokestatic ChangeList.add (I)V
...

protected boolean size_Postcondition(int arg1)
0: invokestatic ChangeList.makeEmpty ()V
...
```

As long as the `add` method is passed a field of the present object, jContractor can identify the fields that are allowed to change by matching the pattern: " `GETFIELD <field>; INVOKESTATIC ChangeList.add;`". It can also detect if `add` is called with an argument that doesn't make sense, as in `ChangeList.add(foo.bar)` or `ChangeList.add("Hello world!")`, and throw an error accordingly. If no ChangeList directives are present in a postcondition, then the method has an implicit change list that allows it to modify all the fields in the object.

A full implementation of the change list concept would require jContractor to monitor the state of fields at run-time, and compare the values before and after method invocation. This can be done, and is the approach that Jass takes. However, creating clones of fields would be computation and time intensive, and would also require that the data types be cloneable. jContractor can determine whether or not a field has the potential to change as a result of method execution before execution begins. jContractor just needs to search the method body for `PUTFIELD` and `INVOKEVIRTUAL` instructions. While this approach does not address the possibility that the field might be modified in an invoked method it is practical and matches the exacting contract specification of the method in hand.

4.4. Performance

Run-time contract checking imposes some performance penalties. Instrumenting class files as they are loaded results in a longer startup time. This time grows linearly with the number of classes that need to be instrumented. Once execution begins, the time required for contract checking grows linearly with the number of contracts that need to be checked.

It is difficult to present meaningful performance metrics for a tool like jContractor since the performance impact of evaluating contracts depends greatly on the nature of the contracts. To illustrate some of the issues involved, we have measured the performance of jContractor running a simple benchmark using the Stack class already discussed. The benchmark pushes 10,000 `java.lang.Integer` objects onto the stack, and then pops them off. Performance was measured on an 863 MHz machine with 128 MB of RAM, running Windows

Table 6. Average execution times for Stack benchmark. All measurements in seconds.

Instrumentation level	None	Pre	Pre & Post	All
Execution time	0.40	1.63	95.68	161.72

Table 7. Average execution times for Stack benchmark without OLD references . All measurements in seconds.

Instrumentation level	None	Pre	Pre & Post	All
Execution time	——	1.74	2.05	2.42

XP, with jContractor instrumenting classes on-the-fly. The results of this test are presented in Table 6.

The benchmark results show that adding contract checking at the precondition level roughly quadruples execution time, in this example. However, when the level of contract checking is increased to preconditions and postconditions, the execution time increases more than two hundred times. The reason for such a drastic increase is the complexity of the contract methods. For example, the benchmark is obtained while evaluating the following contracts:

```
protected boolean push_Postcondition (Object o,
   Void RESULT) {
  return (searchStack(o) != -1) && (size() == OLD.
     size() + 1);
}

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

In order to evaluate the OLD references, jContractor must create a clone of the stack, using the `clone()` method which has complexity of O(n) or worse. The uninstrumented `push()` and `pop()` methods have complexity O(1). In this case we have O(1) complexity functions that become O(n) as a result of contract evaluation.

When the references to OLD are removed from Stack, the execution time decreases drastically, as shown by Table 7. The conclusion to be drawn from this examination is that the performance impact of contract checking depends on the contracts. If the contracts are simple, evaluating them increase execution time, but will not change the time complexity of the program.

5. 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.

5.1. Design By Contract using java

Duncan and Hölzle describe Handshake [8], a dynamically linked library that intercepts JVM's 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 special, Java-like syntax. An advantage of Handshake over jContractor is that it can add contracts to `final` classes and to system classes, whereas jContractor is unable to instrument system classes, due to restrictions in the system class loader. On the downside, Handshake does not support the OLD construct or allow postconditions to refer to the RESULT and needs to be ported to each operating system.

Kramer's iContract [12] 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 convenient syntax, but requires source code availability.

Jass [7] 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, similar to the ChangeList class usage introduced in Section 4.3. Jass also supports "trace assertions," which express constraints on the order in which events occur.

JMSAssert [14], 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.

The Parasoft Corporation produces JContract [25], 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.

Murray and Parson introduce an interesting approach based on using OCL [26] as a specification language to express correctness constraints and the Java Debug Interface (JDI) as a verification API [19]. Contracts are expressed using OCL and fed separately to an auditor application which is in charge of launching the target Java application in a separate VM with appropriate method entry and exit breakpoints enabled. The auditor checks the contraints associated with each entry and exit event and detects violations.

5.2. Introducing design by contract to other languages

Plosch [22] introduce Design by Contract to Python using an approach and syntax similar to iContract. Contracts are specified as assertions in comments by using the keywords req, ensure and inv. A modified interpreter parses the contracts and performs runtime checking of the assertions.

Porat and Fertig propose an extension to C++ class declarations to permit specification of pre- and postconditions and invariants using an assertion-like semantics to support Design by Contract [23].

Cheon and Leavens [4] introduce a runtime assertion checker for the Java Modelling Language (JML) [18]. Their approach is very similar to iContract style of supporting Design by Contract through specifications expressed using a similar special syntax embedded in comments. The drawback of the approach is that it requires specialized JML compiler to produce bytecodes which checks the assertions at runtime.

5.3. Generalized contract specification and instrumentation mechanisms

An example of a specification language for expressing Design by Contract constructs such as invariants, pre- and post-conditions is the object constraint language (OCL) [26], which is a part of the UML standard [21]. OCL allows construction of logical expressions and improves the precision of a UML specification.

Cheesman and Daniels describe in [3] how to specify components in an extended UML version and OCL with contractually specified interfaces. Here the contracts consist of invariants, preconditions, postconditions and intra interface constraints.

Sjorgen describes a method for supporting Design by Contract on the .NET platform using UML and OCL for specifying components in Chapters 2 and 6 of [5].

Nethercote and Seward has developed Valgrind [20] as a meta-tool: a tool for making tools. Valgrind provides a programmable framework for creating program supervision tools for x86 Linux C/C++ developers which can be used for example for debugging, logging, memory leaks and access violations. Valgrind is similar to jContractor in the sense that it performs runtime monitoring based on object code instrumentation. It should be possible to develop generalizable skins to support contract style assertion checks in C/C++ using Valgrind.

Arnout and Simon introduce .NET Contract Wizard [1] to provide .NET developers the ability to add contracts to a .NET assembly independently from the .NET language it is initially written in. This tool takes contracts expressed in Eiffel syntax and in turn creates a proxy to the original assembly (written in Eiffel) without any changes to the original component.

6. 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. K. Arnout and R. Simon, "The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel," in *Proceedings of TOOLS 39, IEEE Computer Society*, 2001.
2. A.P. Black, "Exception Handling: The case against," Technical Report TR-82-01–02, University of Washington Computer Sciences Department, January 1982.
3. J. Cheesman and J. Daniels, *UML Components—A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2000.

4. Y. Cheon and G.T. Leavens, "A runtime assertion checker for the Java Modelling Language (JML)," in *International Conference on Software Engineering Research and Practice (SERP) 2002*, Las Vegas, Nevada, USA, June 24–27, 2002.
5. I. Crnkovic and M. Larsson (ed.), *Building Reliable Component-Based Software Systems*, Artech House Publishers, July 2002.
6. M. Dahm, Byte Code Engineering with the BCEL API, Technical Report B-17-98, Institut für Informatik, Freie Universität Berlin (1998).
7. M.M. Detlef Bartetzko, Clemens Fischer, and H. Wehrheim, "Jass—java with assertions," Vol. 55, 2001.
8. A. Duncan and U. Hölzle, "Adding Contracts to Java with Handshake," Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara (1998).
9. J.B. Goodenough, "Exception Handling: Issues and a proposed notation," *Communications of the ACM*, Vol. 18, No. 12, 1975.
10. C. Hoare, "An Axiomatic basis for computer programming," *Communications of the ACM* Vol. 12, No. 10, 1969.
11. M. Karaorman, 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.
12. R. Kramer, *iContract*—The Java Design by Contract Tool, in J.G. Madhu Singh, Bertrand Meyer and R. Mitchell, (eds.), *Proceedings of TOOLS USA '98, Santa Barbara, California, August 3-7, 1998*, 1998.
13. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading, 1999.
14. Man Machine Systems, "Design by contract for java using JMSAssert," Technical report, URL `http://www.mmsindia.com/DBCForJava.html`
15. B. Meyer, *Eiffel: The Language*, Prentice Hall, New York, 1992.
16. R. Mitchell and J. McKim, *Design by Contract, by Example*, Addison-Wesley, Boston, 2001.
17. B. Meyer, "Object Oriented Software Construction," 2nd ed. Prentice Hall, Upper Saddle River, 1997.
18. G.T. Leavens, A.L. Baker, and C. Ruby, "JML: A notation for detailed design," in H. Kilov, et al., (eds.), *Behavioral Specifications of Businesses and Systems.* Kluwer Academic Publishers*, Boston*, 1999, pp. 175–188.
19. D. Murray and D. Parson, "Automated Debugging in Java using OCL and JDI," in *Proceedings of Fourth International Workshop on Automated Debugging (AADEBUG 2000)*, Munich, August 2000.
20. N. Nethercote and J. Seward, "Valgrind: A program supervision framework," in *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003.
21. Object Management Group, "OMG Unified Modeling Language Specification," report version 1.3, June 1999, Object Management Group, 1999.
22. R. Plosch, "Design by Contract for Python," in *Proceedings of Asic Pacific Software Engineering Conference*, IEEE Computer Society, 1997.
23. S. Porat and P. Fertig, "Class Assertions in C++," *Journal of Object Oriented Programming*, Vol. 8, No. 2, pp. 30–37, 1995.
24. D. Thain and M. Livny, "Error Scope on a Computational Grid: Theory and Practice," in *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, July 2002.
25. "Using Design by Contract to Automate Java Software and Component Testing," Technical report, Parasoft Corporation. URL `http://www.parasoft.com/jsp/products/Jcontract`
26. J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison Wesley, 1999.