Adding Contracts to Java with Handshake

Andrew Duncan and Urs Hölzle
Department of Computer Science
University of California
Santa Barbara, CA 93106
{aduncan,urs}@cs.ucsb.edu
http://www.cs.ucsb.edu/{~aduncan,~urs}

Technical Report TRCS98-32
8 December 1998

Abstract. Contracts describe an agreement between the writer and the user of a class. Their use enhances programmer productivity and program reliability, especially in library code. Handshake provides contracts for Java classes and interfaces in the form of class invariants and method pre- and postconditions. Using Handshake, a programmer can add contracts to classes and interfaces without needing access to their source code, without changing the class files, and without changing the JVM implementation. Unlike existing implementations of contracts for Java, Handshake intercepts the VM's file operations and modifies classes on the fly without requiring any modification to the JVM itself. By using a dynamic link library interposed between the VM and the operating system, the system is relatively simple to port to a new OS and works with a variety of JVM implementations. The system imposes very little overhead other than the time required to evaluate the contract's boolean expressions themselves.

"How joyful am I made by this contract!" — Shakespeare, *First Part of King Henry VI*

1. Introduction

An *assertion* [Ho69] is a boolean expression that must be satisfied for associated code to execute properly. An assertion is not part of the implementation of an algorithm or a class, but describes constraints on the values they use. Programmers use assertions to extend or refine the type constraints already imposed by a class's interface. This combination comprises a *contract* [M88, MM92], executed between a class and its clients (users). In this paper we introduce Handshake, a simple yet flexible way of adding contracts to Java classes and interfaces.

The Java Language Specification (JLS) [GJS96] provides an explicit **interface** type, for describing the types of a class's publicly visible instance methods and class constants. Interfaces express type constraints but not semantic properties expected of a client, nor those guaranteed by the class itself. They cannot specify, for example, that a numeric parameter must be positive, that an instance reference must be non-null, or that a field must stand in some relation to another. Mannion and Philips [MP98] compare this to an electrical outlet: its size and shape constrain the physical properties of any inserted plug, but make no promise about the power provided.

Meyer introduced the notion of “Design by Contract” [M88, MM92] to describe the distribution of responsibility between a class and its users. Methods are equipped with *preconditions* and *postconditions*, and classes themselves with *invariants*. It is the responsibility of a method’s caller to satisfy the precondition; the method itself guarantees the postcondition. Invariants apply to the state of the instance as a whole; they must be satisfied at any time when the object is usable by a client. In order to preserve the contractual semantics of a class in the presence of polymorphism, a class’s assertions must also be combined with those of its ancestors, as described further below.

Programmers have used various approaches to incorporating the advantages of contracts in their Java code. These approaches include explicit tests surrounding each method or additional assertion classes [MP98], preprocessors for augmented Java grammars [K98], and the use of the Java reflection interface [KHB98]. Each approach has its particular shortcomings; among these are the awkward handling of inherited assertions, lack of integration with pre-existing or subsequently-changed libraries, and incompatibility with off-the-shelf Java installations.

Unlike previous implementations of contracts for Java, Handshake uses a *dynamically linked library* to modify the affected classes at runtime. This solution lets the user leave alone the existing off-the-shelf JVM as well as any Java source and class files. A class implementor can provide a separate description of the contract for a Java class, and Handshake’s library then intercedes between the JVM and the operating system, making corresponding modifications to the class file as it is read from disk or across a network, leaving the original unaltered. This approach avoids the problems inherent in other approaches, and provides an effective way of incorporating contracts into working Java code.

Among the advantages of Handshake are:

- It does not modify the Java Virtual Machine’s (JVM’s) semantics or its implementation, so it does not require upgrading or patching when new JVMs are released.
- It can add contracts to Java interfaces as well as to classes, so that abstract data types can be specified without reference to their implementation.
- It does not require source files for the guarded code, so it can add contracts to system or third-party classes.
- It makes no changes to existing class files, so a project developed with Handshake is compatible with any Java implementation.
- It can defer its work until class loading time, so it affects performance only when active.

- It is easy to activate and deactivate the enforcement of contracts. When deactivated, contracts exact no performance penalty.

In the remainder of this paper, we will describe how we fit contracts into the framework of an existing Java installation. Section 2 describes the user's view of Handshake, and discusses alternative implementations. Section 3 defines the semantics of contracts as defined and enforced by Handshake. Section 4 describes our implementation, and section 5 provides performance measurements for real-world classes.

2. Defining and Enforcing Contracts

Handshake provides the tools for describing and implementing contracts for Java classes. A user can specify method preconditions and postconditions and class invariants, and apply them to any Java class. Figure 1 shows how a contract becomes part of a class.

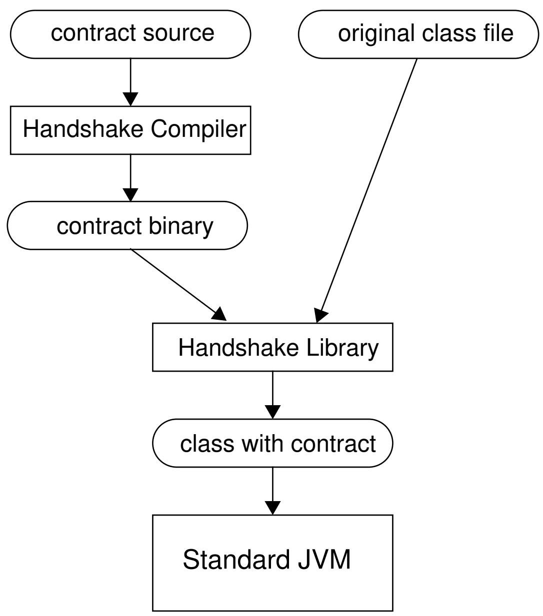

Figure 1. Adding a contract to a class

The contract specification resides in a file separate from the Java class file. A library client could create this file himself, or a library implementor can provide it along with the library class files. The only component of Handshake that the user needs is the Handshake Library and a simple shell script to start the JVM with contracts enabled. Other Java assertion systems do their work before runtime [K98] or after a class has already been loaded [KHB98]; by contrast Handshake works at class load time. When the JVM calls the operating system to request a class file, Handshake intercepts the call and adds assertion code before returning the class. In this way the modification is transparent to both the running virtual machine and the OS: neither knows that it is not interacting directly with the other. Because Handshake operates only at class load time, acts as a logical filter, and leaves no permanent changes, it can be deactivated as easily as it is activated, for example by changing a single environment variable or command line argument. Handshake's implementation as a dynamically linked library has several advantages over other load-time strategies, such as modifying the JVM or creating user-defined class loaders.

In the following sections we describe the process of defining a contract, compiling it into binary form, and enforcing it with the Handshake Library. We also compare our approach to other ways of implementing load-time class modification.

2.1 Defining a Contract

To add a contract to a class with Handshake, a programmer creates a *contract file* which is associated with a single class. Consider a class that implements a stack of integers:

```
class IStack {
   protected int[] _values;
   protected int _size;
   public void push(int i) { ... }
   public int pop() { ... }
   public int top() { ... }
   public boolean full() { ... }
   public boolean empty() { ... }
}
```

The Java type system already provides some form of contract constraining a class's usage. For example, if a client of IStack tries to pass a string to push(), or assign the result of pop() to a character, the Java compiler will catch the mistake at compile time, or the JVM will catch it at runtime. But neither the compiler nor VM can prevent the user from trying to pop from an empty stack. To specify such constraints in Handshake, the stack implementor writes a contract file:

```
contract IStack {
   invariant _size >= 0 "size is negative";

   public void push(int i)
      pre !full() "stack is full" ;
      post top() == i;

   public int pop()
      pre !empty() "stack is empty";
}
```

Each assertion in the contract contains a boolean expression that should evaluate to true. An optional string provides an explanation for an assertion failure.

The syntax of the contract specification is simple:

```
Contract   ::= contract Id '{' Invariant* Clause* '}'
Invariant  ::= invariant Expr [String] ';'
Clause     ::= MethodHeader Pre* Post*
Pre        ::= pre Expr [String] ';'
Post       ::= post Expr [String] ';'
```

Here we use the BNF notation of square brackets for optional constructions; the Kleene star has its usual "zero-or-more" meaning. The non-terminal MethodHeader is the same as specified in the JLS §19.8.3. The non-terminal Expr is essentially the Primary expression type from the JLS §19.12, excluding instance and array creation and assignment. (The excluded expressions have side-effects, so they are not necessary in assertions.)

Figure 2 shows the changes that Handshake will apply to the IStack class to implement its contract.

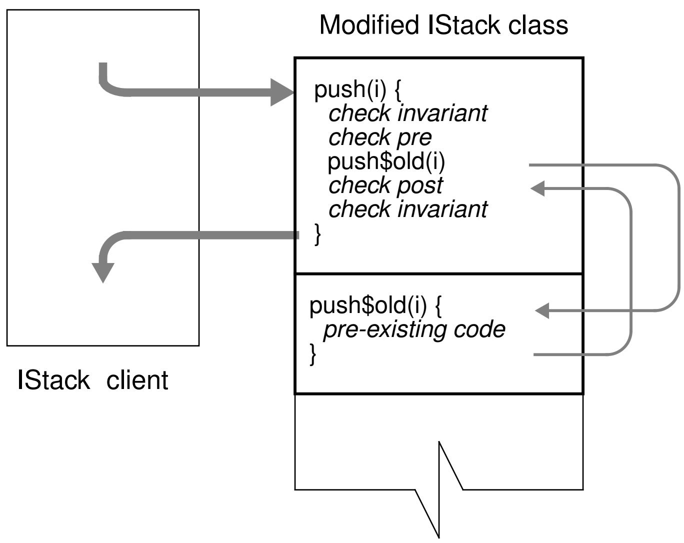

Figure 2. Renaming and wrapping the `push()` method

With contracts enabled, the actual code generated for the new push() method will be:

```
public void push(int i) {
    if (!(_size >= 0)) throw new RuntimeException("Invariant failed: size is negative entering IStack.push().");
    if (!(!full())) throw new RuntimeException("Precondition failed: stack is full in IStack.push().");
    push$old(i);  // Call original version.
    if (!(top() == i)) throw new RuntimeException("Postcondition \"top() == i\" failed in IStack.push().");
    if (!(_size >= 0)) throw new RuntimeException("Invariant failed: size is negative leaving IStack.push().");
}
```

For each method named in the contract, Handshake renames it, constructs a new method that evaluates the appropriate assertions, and makes a nested call to the renamed method. If an assertion does not hold, the added code throws an exception with the appropriate message from the contract specification.

2.2 Enforcing a Contract

At class load time, the Handshake Library combines the contract file with the original class file to produce the class with its contract. The library acts as a simple filter between the Java runtime system and the OS-dependent libraries used by the VM. The same Handshake Library can implement contracts for a variety of unmodified Java virtual machines that share the same operating system libraries. Figure 3 shows a simplified picture of the data path for this process.

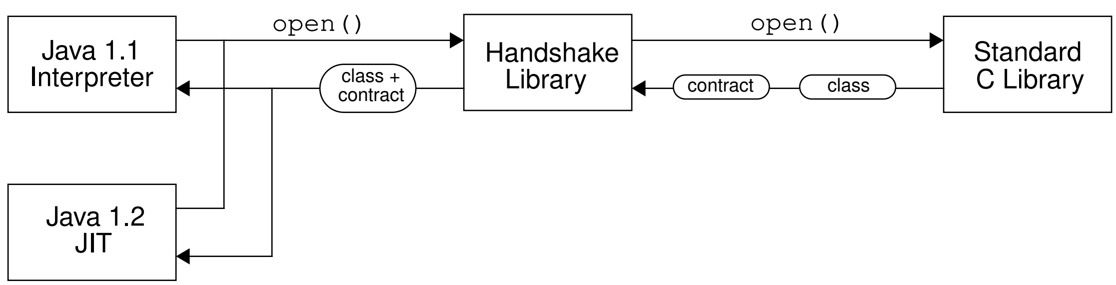

Figure 3. Intercepting a system call with the Handshake Library

To activate the assertions in the contract, the user arranges for the JVM to use the Handshake Library, which is an augmented version of the dynamically linked standard C library. The Handshake Library intercepts the JVM's calls to open class files, and provides it with modified classes, adding the assertions defined in the contract. The original class file, on disk or across a network, does not change. In the event that the user wants to discontinue use of the contract, there are no complex binary edits to undo.

Handshake will look for the associated contract file in the same directory as its class file, or in a user-specifiable repository directory, and apply its changes to the class file. A class implementor can provide the contract file along with its class file; clients need not concern themselves with the process of compiling the contract, just with reading and understanding it.

A convenient feature of this approach is that it is robust with respect to changes in the code being guarded with assertions. As long as the interface of a guarded class does not change, the assertion-adding process will continue to work correctly. In other words, if a class changes during development, no explicit programmer action is needed to continue enforcing contracts. Even if parts of the interface do change, Handshake can still add assertions to the unchanged parts.

2.3 Alternative Implementations

The Handshake Library functions as a filter between the JVM and the operating system libraries. Alternative ways of intervening at the same juncture include modifying the JVM's file-opening calls directly or writing a class loader in Java. In addition, a user could add contracts to class files off-line, that is, before runtime. However, each of these possible approaches has its shortcomings.

An obvious way of providing contract functionality is to integrate it into a JVM. It is simple to modify the JVM's file-handling code directly, adding a stage that looks for contracts and adds them when reading class files. However, modifying the JVM creates a significant restriction: contracts will only work with a particular VM. Very few Java virtual machines have publicly available source code, so adding support to all popular VMs is not an option. By contrast, Handshake can work with off-the-shelf JVMs and only needs one port per OS. For each new VM, only a small amount of work is necessary (writing or modifying a short shell script).

The Java DevelopmentKit (JDK) 1.2 specification includes an extension mechanism that allows users to define their own class loaders [LB98]. Class loaders are a consistent and portable scheme for modifying classes during loading, but they have several shortcomings for the task at hand. First, they apply only to user-defined classes; all system classes (*e.g.* java.lang.*) are loaded directly by the JVM, so this approach could not add assertions to such classes. Second, applications may use their own class loaders, bypassing the loader that inserts the contract assertions. Finally, user-defined class loaders influence the semantics of class accessibility [LB98]. In an application with two class loaders, both of which delegate to the contract-adding class loader, class namespaces that were formerly separate will merge, with unpredictable results.

Another approach would modify classes on disk, creating a parallel set of classes. In this case, the user would run a tool adding the contract code after every recompile. But this is added work for the programmer, who has to keep track of out-of-date classes and contracts, and maintain a separate repository of modified classes. The Handshake system achieves the same goals in an automated, incremental, and transparent way, checking cached classes at each access and rebuilding them as necessary.

3. Semantics of Handshake Contracts

A contract augments a class's interface with constraints on its state as a whole and on entry and exit conditions for its methods. (This excludes from a contract such assertions as *loop invariants* [M88], or other constraints on the values variables may have inside a body of code.) Handshake provides three kinds of assertions that can be associated with a Java class or interface: method preconditions and postconditions, and class invariants. Because our focus was to find effective and practical ways to integrate contracts into standard off-the-shelf Java Virtual Machines, Handshake currently takes a conservative approach (modeled on Eiffel contracts), implementing only the essential features of contracts without adding extra bells and whistles.

Handshake assertions consist of a keyword, an optional string, and a boolean expression. Syntactically the expression may be any legal Java expression except instance creation, array creation, or assignment. The expression may refer to any member normally visible on entry to the guarded method.

A precondition specifies a predicate that must hold on entry to a method; if it is violated, the caller of the method is responsible. Postconditions describe the guarantees a method makes, provided its preconditions are met; if a postcondition fails there is a bug in the method itself. Postconditions for non-**void** methods may refer to the variable $result, which is set to the value the method will return. If a method exits because of an uncaught exception, Handshake makes no effort to check the postconditions. Finally, an invariant describes properties that must hold for the object to be in a consistent state and to behave properly. Handshake will evaluate invariants at entry (before any preconditions) and exit (after postconditions) of any non-**private** method listed in the contract.

Contracts extend the semantics of a class's type, and just as substitutability of derived classes requires type conformance, it also requires contract conformance. Thus, preconditions are *contravariant* with inheritance, whereas postconditions are *covariant*. (This situation is directly analogous to the rules for substitutability of types [SOM93].) Handshake constructs proper variance by logically or-ing an overridden method's preconditions with those of its ancestor; the postconditions will be and-ed.

Handshake considers a violated contract to invalidate all further execution of the guarded method, and immediately throws a java.lang.RuntimeException. Currently, Handshake does not provide specialized subclasses of this system class, so that projects using contracts will not need additional classes.

4. Implementing Handshake

The Handshake system consists of two parts: a compiler and a dynamically linked library. The compiler is written in Java using compiler construction tools and invokes the Java compiler itself to generate bytecodes for the assertion expressions. The library is an edited and re-linked version of the standard dynamically linked C library used by the JVM.

4.1 The Handshake Compiler

The Handshake Compiler has two stages: the contract compiler and the BCA compiler, as shown in Figure 4. The overall result of running the Handshake Compiler is to translate a contract into a binary file that is easy to combine with a class file at load time. The Handshake Compiler is written in Java and contains about 130 classes.

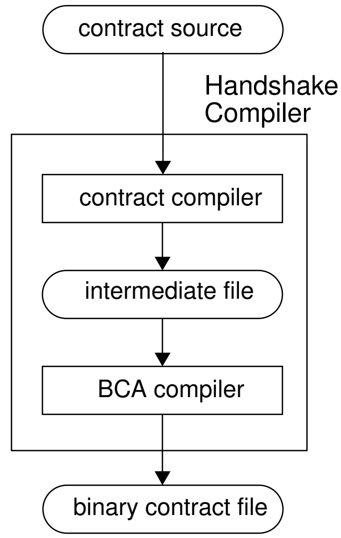

Figure 4. Components of the Handshake Compiler

4.1.1 The Contract Compiler

The contract compiler is a text-to-text translator that processes a contract file into an intermediate text file, which is the input to the second stage of the Handshake Compiler. Using this step is not necessary, but the text form of this file is easy to read and understand, provides a convenient debugging checkpoint, and allows us to reuse an existing tool for manipulating class files.

We built the contract compiler by combining the contract syntax with an existing LALR grammar for Java expressions, and feeding the result to the compiler tools JLex [B96] and JavaCUP [Hu96] to create a parser and syntax tree builder for the contract language. Because the tree is available for inspection, and supports the Visitor design pattern [GOF95], it is relatively easy to extend the functionality of the contract compiler to support additional features of contracts.

The resulting intermediate file provides a more fine-grained language for expressing an explicit description of the changes to be made to the class file at load time. The syntax and semantics of this language as well as details of its compiler are defined in [KH98]. Briefly, this file can specify renaming class or instance members or references to them, or adding members, and carrying out these changes to specific classes or interfaces, or all implementors of an interface. Thus the language provides a highly flexible internal path for the current Handshake implementation, and allows for future expansion. The intermediate file corresponding to the example contract for the IStack class has this form:

```
delta IStack adapts class Istack {
   rename method void push(int i) to push$old;
   add method public void push(int i) {
      if (!(_size >= 0)) throw new RuntimeException("Invariant failed: size is negative entering IStack.push().");
      if (!(!full())) ...      // Similar, as in section 2.1.
      push$old(i);        // Call original version.
      if (!(top() == i)) ...
      if (!(_size >= 0)) ...
   };
   rename method int pop() to pop$old;
   add method public int pop() { ... };
}
```

As the file shows, when Handshake adds an assertion to a method, it first changes the method's name by appending the characters $old. As per the JLS §3.8 the '$' character is to be used only for mechanically generated code; this convention should minimize the possibility of name clashes. Handshake then defines a new method with the original name. Calls to the original method will thus be redirected to the new one. The new method contains the code for testing the assertions and throwing the appropriate exceptions.

Since Handshake uses a wrapper method around the original guarded method, multiple exit points to a method pose no problem. If the method returns a value (*i.e.* is non-**void**) Handshake generates code that stores it in the local variable $result pending the postcondition's evaluation. The wrapper method does not currently check for thrown exceptions, so if the code in the original method throws an exception, the postconditions and invariants will be skipped.[1]

In the interest of simplicity, Handshake currently requires one contract file per class. However Java supports the combination of many class files into a single *jar file*. It would not be difficult to extend Handshake in a similar way, for example by allowing one compound contract file per jar file.

4.1.2 The BCA Compiler

The BCA compiler translates the intermediate specification into a more compact representation, the binary contract file, containing Java bytecodes and associated annotations. This format allows for efficient incorporation into an existing class file at load time to produce the modified version. The details of the compiler and the binary file's structure are in [KH98].

4.2 The Handshake Library

On virtually all operating systems, programs use dynamically linked libraries to insulate themselves from details of the operating system and hardware they are running on. For example, such libraries provide the code for familiar file-handling functions like open(), close(), read(), and so on. The Handshake Library is a

[1] We are currently considering supporting postconditions for exception returns in the spirit of Karaorman's proposal [KHB98].

modified version of the dynamically linked standard C library. We change this library in a manner similar to the way Handshake changes Java classes: that is, we add a wrapper to the functions that open a file or return information about it. For example, when the library is called to open a Java class file, it does its own processing, in the midst of which it calls the C library file functions itself.

Figure 5 shows some of the control and data flow for process of adding a contract to a class. In the figure, the Java Virtual Machine (shown at left) calls the open() library routine to start the loading of a Java class. Handshake's version of open() reads the class file and the contract file by calling the C library's original open() function. Handshake parses the class file into an internal representation, applies the changes specified in the contract, translates the result back into class file format, and writes the modified class file to a user-specifiable cache location on disk. Finally, Handshake passes the new path to the C library's file-opening routine and returns the resulting file descriptor.

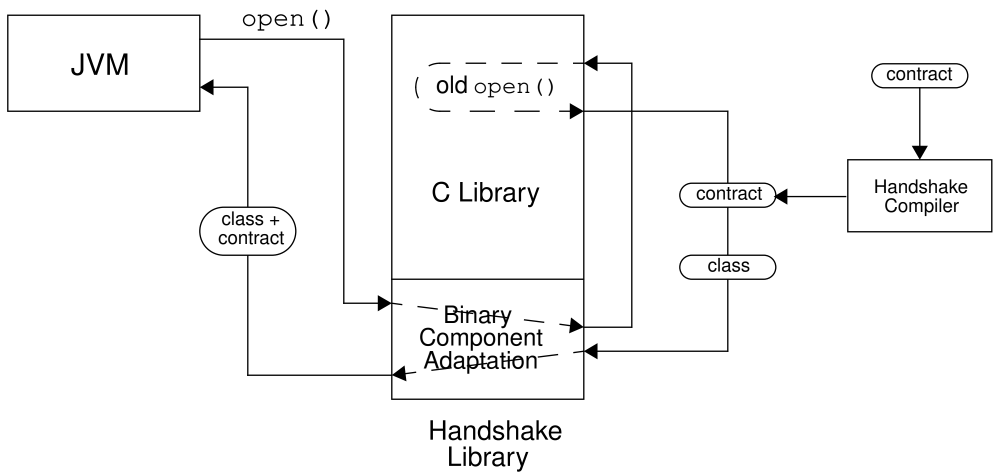

Figure 5. Data and control flow in the Handshake system

When the Handshake Library detects a request to open a Java class file, it first looks in its cache directory to see if the class has already been modified and stored. If so, Handshake just redirects the system call to the cached version and passes the call to the C library. If not, Handshake looks for a contract file first in the same directory as the class file, then in the cache directory. Looking in the cache directory allows for specifying contracts for classes (*e.g.* system classes) whose directories are not writable by the Handshake user. If there is a contract file for the class, Handshake applies the changes to the class file, caches it and redirects the system call to the new file. If there is no contract file, Handshake has no effect, just passing on the call unchanged to the C library. When Handshake locates a cached version of the class file, it also checks to see that the class file is up-to-date with respect to its contract file and the original class file. If the class file is out of date, the library reapplies the contract file to the class and caches the result as described above.

The standard C library implements many file-related routines. For example, the Solaris 2.6 library libc.so.1 contains functions open(), _open(), and several others. To make the Handshake Library work with all JVMs on a given platform, we modify each file routine to check for class files. In addition to open() and variants, we also instrument the stat() call which returns information about the file, in particular its size. In all functions the changed versions go through the process described above, looking for the contract file and modifying the class file if appropriate.

The procedure for creating the Handshake Library is similar for most platforms: edit an existing library, compile replacement code, and link them together. For example, to create a version for a Unix platform, we first make a working duplicate of the dynamically linked C library libc.so.1. We then edit this binary file directly, renaming the functions we want to intercept, *e.g.* open() becomes OPEN(). We create a new source file redefining open() to carry out the process of adding contracts to classes, itself calling the renamed C library version as necessary. Handshake also incorporates part of the BCA runtime library [KH97, KH98] to perform class file modifications. Finally, we compile and link this code with the existing C library, resulting in the Handshake Library.

Because the Handshake Library is dynamically linked, it suffices to arrange that the off-the-shelf JVM finds and uses it in place of the standard C library. Usually, a simple shell script setting the library search before starting the JVM is all that is needed to add contracts to a new JVM.

Our current implementation for Solaris 2.6 supports all Solaris JVMs (*e.g.* the JDK 1.1.x interpreters, the JDK 1.2 VM, and the Solaris 1.2beta4 JIT); these JVMs differ in the way they open class files but are all supported by the same library. We are currently porting our code to the Win32 platform and its popular JVMs.

4.3 Performance

To evaluate the overhead imposed by contracts, we added contracts to 107 classes of javac, the Java compiler provided in the JDK. Table 1 shows the times required for various combinations of libraries, contracts, and caching policies. We measured the performance using JDK 1.2beta4, on a Sun Ultra-1/170 167MHz workstation running the Solaris 2.6 operating system.

The first line of the table shows the time required for javac to compile a “Hello World” program. In this case the compiler did not use the Handshake Library. The second line shows the added time required when the compiler’s file-handling system calls went through the Handshake Library. The overhead—less than half a second—takes place when the compiler asks for class files: Handshake looks for a cached class file and for a contract file (in this case unsuccessfully) before returning the originally requested file.

Library and contract type	User time (s)	Kernel time (s)	Total CPU time (s)	Total running time (s)
Unmodified program	2.63	0.50	3.13	3.42
Handshake Library, no contracts	2.78	0.71	3.49	3.96
Handshake Library, 107 contracts, cached classes	2.87	0.63	3.50	3.92
Handshake Library, 107 contracts, no cached classes	3.49	0.78	4.27	4.77

Table 1. Overhead of Handshake instrumentation

The third line represents a typical execution of a program instrumented with contracts. We added empty contracts to 107 classes of javac, executed it in order to generate the cached classes, and then measured the overhead of using the cached classes. The overhead here is similar to that with no contracts; in this case the search finds a cached file and returns it instead of the original one.

The fourth line shows the time required to run the compiler from a "cold start," with no classes cached. In this case Handshake has to find the contract file, ascertain that there is no cached class, read and parse the class into memory, perform the modifications (in this case none), un-parse and write the file to the cache and return that file. Even in this case the performance penalty for total running time is only 1.3 seconds, or about 1/100th of a second per instrumented class.

5. Related Work

Eiffel [M91] incorporates contracts as an integral part of a class's interface. It supports pre- and postconditions, class invariants, loop variants and invariants, and mid-method checks. Eiffel automates the inheritance of assertions for overridden methods, and provides a mechanism for retrying code that triggers an assertion violation and for restoring an instance's state to that before the violation. It also allows a wide range of control over which assertions or classes of assertions are to be tested at runtime.

One way to express and enforce contract constraints is to include in code explicit tests for the desired properties. Mannion & Philips [MP98] discuss the elementary addition of assertion classes with static methods for testing assertions and throwing exceptions. However, the programmer must have access to the program source to add them. Calling these static methods at entry and exit to a method inflates its body with a considerable amount of code that does not conceptually belong there—that is, it does not implement any of the method's semantics. Such assertions are harder to distinguish from implementation code, and so cannot be used by automated tools to reason about a program's properties. It is also difficult to enable or disable assertions selectively with this scheme. Moreover, simple predicate tests are insufficient to handle correctly the interaction of assertions with inheritance. The user can add additional pre- and post-condition wrapper methods for every guarded method, but this is awkward and error-prone. Instead of inflating the individual methods, we now have three or more methods where one existed before.

Kramer's iContract [K98] specifies an extension to the javadoc comment syntax, and provides a preprocessor that converts the assertion comments into code. The advantage of this system is that the contract is to some extent part of the official interface of the class, as extracted by the standard JDK tool javadoc. In addition, iContract supports limited versions of existential and universal quantifiers, adds code that tests for infinite recursion in assertion evaluation, and avoids testing invariants on intra-instance calls (sends to self). However, this approach requires access to the source file; in many cases this may be impractical or impossible.

Payne, Schatz, and Schmid describe an assertion class that makes the expression of assertions and of the control flow of violations easier [PSS98]. Their company, Reliable Software Technologies, sells a commercial product that also defines an extension of the Java comment syntax, and uses the extended comments to instrument class files with assertions.

Karaorman's jContractor [KHB98] uses the reflective capabilities of Java and the Factory Method design pattern [GOF95] to support the addition of contracts. When objects are instantiated through a special factory class, the factory adds to the new instance the instrumentation code. This approach uses only existing, standard Java constructs and calls. It also provides for retrying a method that throws an exception, and rolling back variables to earlier values. However, the user must specify as distinct private boolean methods each precondition, postcondition, invariant, and exception handler, leading to a great proliferation of class members.

Another way to intercept system calls is by using the /proc file system available on some versions of the Unix operating system [FG91, AISS98]. Files under this directory are virtual files that represent the address spaces of running processes. In this approach, the JVM runs as a child process. The parent process uses the ioctl() call to intercept input and output to the child's virtual file, which are equivalent to system calls by the process. This approach is limited to Unix environments. Other problems include that the child process runs in a different address space from that of the parent, and has a different space of file descriptor indices.

6. Future Work

We designed Handshake to be simple to use. If a system does not satisfy this criterion, no longer list of features will convince developers to use it regularly. Handshake already supports the core ideas of contractual design. However, there are some features that we may add to Handshake contracts, for example support for the **old** keyword in postconditions, to express the state of the instance on entry to the method; greater selectivity in activating or deactivating assertions; or not checking invariants in nested method calls within an instance.

Handshake currently operates only on individual class files and does not support jar files, Java libraries stored in a compressed format, many classes to a single jar file. We will add support for jar files in a future version by uncompressing the classes, adding contracts to those classes with associated contract files, and recompressing the result before passing the jar file along to the JVM.

Handshake does not currently manage package namespaces in its cache directory; we are currently adding this functionality.

7. Conclusion

We have designed and implemented a novel way to add language-like functionality to Java implementations without needing to change the JVM implementation, the class files, or the source. Once the Handshake library has been ported to a new operating system, we can add contract support to a new off-the-shelf JVM with little effort, usually by creating or modifying a simple script. Thus, even without having privileged access to JVM sources, we can provide uniform support for contracts across many different Java implementations.

In addition to this portability advantage, Handshake also allows the programmer to add contract assertions to existing classes (even system classes) and makes the process of activating these contracts completely transparent—the programmer never has to worry about re-running preprocessors are any other tools. Furthermore, Handshake allows programmer to add contracts to Java interfaces (not just classes), thus considerably boosting the usefulness of contracts.

Handshake exacts only a modest performance penalty for using contracts. The only overhead typically comes from an additional file lookup each time the running program asks for a class. This extra time represents a small fraction of the total running time of any substantial program.

Contracts are an important generalization of the strongly-typed interfaces with subtyping that Java and other object-oriented languages support. We believe the Handshake system is a practical and effective way to provide the advantages of contracts to Java developers and users.

Acknowledgments

This work was funded in part by Sun Microsystems, the State of California MICRO program, and by the National Science Foundation under CAREER grant CCR96-24458 and NSF grant CCR-9804061. Many thanks to Ralph Keller for his extensive work on binary component adaptation. We are also grateful to Albert Alexandrov for his discussions on intercepting system calls, and to Murat Karaorman for his advice about contracts and assertions.

8. References

[AISS98] Albert Alexandrov, Maximilien Ibel, Klaus Schauser, and Chris Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. In *Proceedings of the USENIX 1997 technical Conference*, Anaheim, CA, January 1997, pp. 77-90.

[B96] Elliot Berk. JLex, a Lexical Analyzer Generator for Java. Available at http://www.cs.princeton.edu/~appel/modern/java/JLex.

[FG91] R. Faulkner and R. Gomes. The Process File System and Process Model in UNIX System V. *Proc. of the Winter 1991 USENIX Technical Conference*, January 1991.

[GOF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Software*. Addison-Wesley, 1995.

[Ho69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), October 1969.

[Hu96] Scott Hudson. JavaCUP, a Parser Generator for Java. Available at http://www.cs.princeton.edu/~appel/modern/java/CUP.

[K98] Reto Kramer. iContract—The Java™ Design by Contract™ Tool. Available at http://www.promigos.ch/kramer.

[KH97] Ralph Keller and Urs Hölzle. Binary Component Adaptation. *Proc. of the European Conf. on Object-Oriented Programming*, Springer-Verlag, July 1998.

[KH98] Ralph Keller and Urs Hölzle. Implementing Binary Component Adaptation for Java. Technical Report TRCS98-21, University of California, Santa Barbara, 1998.

[KHB98] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: Reflective Java Library to Support Design-by-Contract. Technical Report TRCS98-31, University of California, Santa Barbara, 1998.

[LB98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, October 1998.

[M88] Bertrand Meyer. *Object-Oriented Software Construction (1st Ed.)*. Prentice-Hall, 1988.

[M91] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

[MM92] Bertrand Meyer and Dino Mandriolo (ed.). *Advances in Object-Oriented Software Engineering*. Prentice-Hall, 1992.

[MP98] Mike Mannion and Roy Phillips. Prevention is Better Than a Cure. *Java Report*, September 1998.

[PSS98] Jeffery E. Payne, Michael A. Schatz, and Matthew N. Schmid. Implementing Assertions for Java. *Dr. Dobb's Journal*, vol. 23, no. 1, January 1998.

[SOM93] Clemens Szypersky, Stephen Omohundro, and Stephan Murer. Engineering a Programming Language: the Type and Class System of Sather. ICSI Tech Report Tr-93-064, University of California, Berkeley, 1993.