

## Supporting Design by Contract in Java

**Martin Lackner**

Decision Management Systems  
Wallnerstraße 2, A-1010 Wien, Austria

**Andreas Krall, Franz Puntigam**

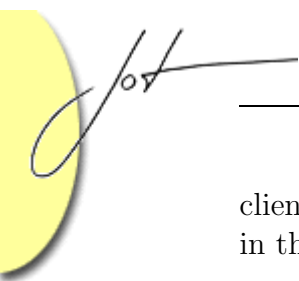
Technische Universität Wien, Institut für Computersprachen  
Argentinierstraße 8, A-1040 Wien, Austria

Design by Contract is a valuable design method for trusted software components. Eiffel shows how to provide appropriate language support for it. However, no such concepts currently exist in Java. Full integration of them into Java may help to improve and guarantee the quality of Java classes. We briefly compare several approaches to extend Java in this way and present our model and a compiler that translates extended Java code into JVM byte code. Our Java extension integrates preconditions, postconditions, and invariants as in Eiffel while respecting the characteristics of Java. The evaluation shows that Design by Contract can be added efficiently to Java while keeping compatibility.

### 1 INTRODUCTION

Java has become the dominant language in the area of Internet computing and is also very successful in other domains. Not only because of its strong type concept Java seems to be quite appropriate for the development of high quality software components. However, Java does not support some well-explored language features highly important to software of guaranteed quality. One of them is genericity. In the present paper we deal with another one, the missing support of Design by Contract.

Bertrand Meyer introduced Design by Contract [Mey92, Mey98] in Eiffel [Mey91] as a powerful technique for reliable software. Its key elements are assertions. These are boolean expressions that define correct program states at arbitrary locations in the code. Simple assertions belong to individual statements. Assertions belonging to interfaces are more important: Preconditions have to be satisfied on method invocation, postconditions are checked after method execution, and invariants express conditions for the consistency of data. These assertions in a type or class specify a contract between its instances and their clients. According to the contract a server promises to return results satisfying the postconditions if a client requests available services and provides input data satisfying the preconditions. Invariants must be satisfied before and after each service. Neither party to the contract (these are the



clients and servers) shall be allowed to rely on something else than stated explicitly in the contract. Some important goals of Design by Contract are

- reduced test effort by separating contract checks from regular application logic,
- saved debugging effort due to improved monitoring where failures occur close to the faults,
- as well as improved up-to-date and unambiguous documentation of interfaces.

In an old version of the Java specification [Gos94], the last one James Gosling wrote alone, assertions were part of it. Because of a deadline Gosling removed assertions from the specification. Recently Sun added a simple assertion facility into Java (JDK 1.4) using the new keyword `assert`. There is still no direct support of preconditions, postconditions and invariants.

It is possible to simulate preconditions, postconditions, and invariants by using conventional programming techniques. For example, the code for postcondition checking can simply be added to the end of a method. However, the goals of reduced test effort and up-to-date documentation cannot be achieved because of missing separation of contract checks from regular application logic. We need special language support to meet these goals.

The rest of this paper is structured as follows. In Sect. 2 we give a brief overview of several approaches to extend Java with assertions. In Sect. 3 we present our own proposal to extend Java with preconditions, postconditions, and invariants. Next, in Sect. 4 we discuss some implementation details of the proposed extensions. Finally, in Sect. 5 and Sect. 6 we show performance results and draw our conclusions.

## 2 APPROACHES TO DESIGN BY CONTRACT IN JAVA

Most proposals use preprocessing techniques instead of a full language integration. We discuss some of them.

### Jass

Jass [Ber99] supports different kinds of assertions like pre- and postconditions, class invariants, loop invariants, and checks at arbitrary locations in the code. Expressions in assertions must be purely declarative; no side effects such as assignment and instance creation are allowed. The programmer can use old field values and method results in postconditions. Fig. 1 shows an example of assertions in Jass. In the postcondition, `Old.a` is the old value of the field `a`. If an old value is used, the object is cloned at the beginning of the method. In the postcondition, the field value is compared to its counterpart in the clone.



```

public class A implements Cloneable {
    protected int a;
    public void addValue (int x) {
        /** require x > 0; */
        a = a + x;
        /** ensure Old.a > a; a > 0; */
    }
    protected Object clone() { ... }
    /** invariant a > 0; */
}

```

Figure 1: Conditions in Jass

If the evaluation of an assertion invokes a method that declares another assertion, the second one should not be evaluated. Jass partially implements this principle. The preprocessor introduces copies without any assertion checks for methods called by assertions in the same class. These copies are invoked within assertions. However, the principle is violated if methods call methods from other classes.

Subclasses must not refine the conditions in parent classes. A programmer who needs refinements has to implement the interface `jass.runtime.Refinement` used as a signal to the preprocessor. Then, the preprocessor copies all assertions from the parent class into the subclass. This approach has some disadvantages: Private variables are not allowed to occur in invariants because (copied) invariants in a subclass cannot access them. Furthermore, invariants in base classes may not be checked in overridden methods. No invariants are checked on exceptional termination.

Jass extends the Java exception model with the rescue- and retry-mechanism of Eiffel. A new version of Jass supports trace assertions [Pla00]. The programmer defines allowed sequences of method calls that are dynamically compared with the real call trace.

## iContract

iContract [Kra98] also uses a preprocessing technique: It copies the parent class into the subclass. If the source code of the parent class is not available, the preprocessor produces a special repository class. The syntax of iContract complies with UML/OCL [WK99] (see Fig. 2). Invariants are not enforced for private methods because they already were enforced upon invocation of the public method preceding the private one. To automatically avoid non-terminating recursion, iContract instruments check code such that it keeps track of the call chain at run time. This mechanism is thread-safe. Invariants, preconditions, and postconditions can refer only to non-private instance variables unless the class is final. If an expression *expr* is of the type `Cloneable` or `String`, the value of *expr@pre* is a shallow copy of

```

/**
 * @pre  o != null;
 * @post list.size() == list.size()@pre + 1;
 */
void append(Vector list, Object o) { ... }

```

Figure 2: Pre- and postconditions (with old value) in iContract

*expr*; otherwise *expr*@pre and *expr* are identical. Unlike Jass, iContract clones only necessary object parts.

The features of JMSAssert [Ran00] are very similar to those of iContract. However, JMSAssert has better performance because it adds a library to the JVM so that the virtual machine supports assertions directly.

## jContractor

jContractor [KHB98] is a Java library and a design-pattern-based approach to support Design by Contract. The programmer adds the contract code to a class in the form of methods as shown in Fig. 3.

```

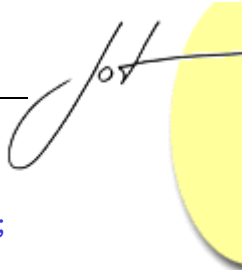
class ClassName ... {
    Object methodName(Object x, String key) {
        /* method body */
    }
    protected boolean methodName_PreCondition(Object x, String key) {
        /* Precondition of above method */
    }
    protected boolean methodName_PostCondition(Object x, String key) {
        /* Postcondition of above method */
    }
    protected boolean className_ClassInvariant() {
        /* Class invariant */
    }
}

```

Figure 3: Conditions in jContractor

A pattern defines the signature of these methods. All methods use a special naming convention and are protected. The class loader looks for these patterns and rewrites the code of the loaded class. jContractor supports old values of object fields in postconditions as in this example (*OLD* is a class with a static method *value*):

```
count == OLD.value(count) + 1;
```



```

class IStack {
    protected int values[], size;
    public void push(int i) {...}
    public int pop() {...}
    public int top() {...}
    public boolean full() {...}
    public boolean empty() {...}
}

contract IStack {
    invariant size >= 0 "size < 0";
    public void push (int i)
        pre !full() "stack full";
        post top() == i;
    public int pop()
        pre !empty() "stack empty";
}

```

Figure 4: Stack class and contract file for Handshake

The class loader replaces all `OLD.value(...)` expressions by simple variable expressions referring to the temporary variables used to record the method entry value of the attribute.

## Handshake

Handshake [DH98] employs a dynamically linked library and works by intercepting JVM's file accesses and instrumenting classes on the fly by using a mechanism called *binary component adaptation*. Handshake allows a programmer to write external contract specifications for Java classes and interfaces without changing the classes themselves as shown in Fig. 4. Binary component adaptation has been developed for on-the-fly modification of precompiled-Java components (class byte-codes) using external specification code containing directives to alter the pre-compiled semantics. A drawback of this approach is the necessity of external contract specifications in a special syntax and the Handshake library as a non-Java system that has to be ported to different platforms.

Handshake does not yet support old values in postconditions. The result value is available as the special variable `$result`.

## 3 OUR LANGUAGE EXTENSIONS

Most proposals translate the source code into an instrumented source code. This approach violates abstraction and modularity. Source-to-source translators like Jass and iContract make debugging of the byte code very uncomfortable and prevent separate compilation because the source code must be available to the compiler. Method removal at load time as in jContractor may confuse the programmer. All proposals except jContractor put assertions in special Java comments.

Our proposed extension adds new keywords to Java so that the programmer can define class and method constraints as part of the code. We do not place pre- and postconditions in comments because of possible side effects. The proposed syntax

is upward compatible to Java. We make heavy use of the simple assertion facility added to Java in JDK 1.4. We need not change the class loader or access the source code of parent classes. Abstraction, modularity and separate compilation are fully supported. The code can be translated to Java byte code in one step, making debugging easier than precompiler-based approaches.

## Syntax and Semantics

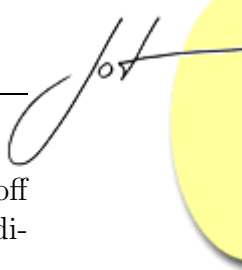
Three keywords are added to the syntax: `@invariant`, `@require`, and `@ensure`. In each class or interface, a single *class invariant* can be defined. It starts with `@invariant` and contains an arbitrary number of assertions (see Fig. 5). Invariants are checked at the entry and at the normal or exceptional return from a non-private method. However, invariants need not hold if constructors throw exceptions because the exceptions' reasons may prevent class initializations that comply with the invariants. If an assertion in an invariant fails, the evaluation of the invariant ends and an `InvariantError` is thrown. Invariants need not be enforced for private methods because they were already enforced for the exported method called prior to the private one. In fact it is desirable to allow private methods to temporarily violate class invariants.

```
class Account {
    @invariant {...} // class invariant

    public int getDeposit()
        @require {...} //preconditions
        {
            ... // method body
        }

    public int withdraw (int withdrawal)
        @require {
            assert (withdrawal >= 0);
            ... // preconditions
        }
        {
            ... // method body
        }
        @ensure {
            ... // postconditions
        }
}
```

Figure 5: Example of class invariant, pre- and postconditions



According to JDK 1.4 assertion checking in a class can be switched on or off before loading the class. When assertions are switched off, the invariants, preconditions, and postconditions in the class shall not be checked at all.

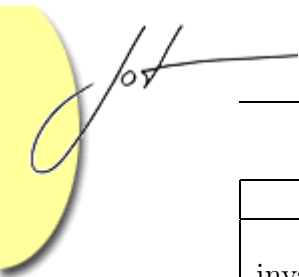
Preconditions are defined at the beginning of methods directly after formal parameter lists. They start with the keyword `@require` and contain assertions (see Fig. 5). It is the duty of callers to satisfy preconditions at method entries. Some preconditions of public methods shall be checked inside methods that throw exceptions like `IllegalArgumentException` and `IllegalStateException` to ensure that necessary requirements are fulfilled even if assertions are disabled. If the `@require` clause fails, a `PreconditionError` exceptionally terminates the method.

Postconditions are located directly after method bodies and start with `@ensure`. The assertions are checked prior to method returns immediately before checking class invariants. Failed assertions throw a `PostconditionError` exception. Postconditions are not checked for methods returning an exception.

Local variables defined in method bodies must not be used in the pre- and postconditions. Thus, `@require` clauses are located before and `@ensure` clauses after method bodies. Assertions in these clauses can refer to private fields of the class. Eiffel prohibits access to private fields. We decided to allow these accesses for practical reasons. For example, if we read a private variable instead of using a public getter method, the semantics of the assertion remains unchanged when the getter method is overridden. We may use this assertion to ensure that overridden getter methods always return a value not larger than the value in the private variable. Accessing private variables should be used with care because it may make it harder to understand the conditions for method invocations.

If the value of a public field of an instance is changed, the invariant of that instance can fail. A correct evaluation of the precondition is only guaranteed if the invariant is satisfied. Therefore, invariants are checked before preconditions. Postconditions are checked before invariants, but invariants are also evaluated in the case of exceptional method termination. Failing invariant checks absorb failing postcondition checks. Synchronized methods acquire their locks before checking invariants and preconditions and release them after checking postconditions and invariants (again).

In general, preconditions are checked on method entry, postconditions on normal method return, and invariants on method entry as well as on normal and exceptional method return. Table 1 summarizes these rules. Pre- and postconditions can be defined for static methods. However, invariants are not checked in static methods and static initializers because access to instance variables is not allowed in static methods. So far we avoided to introduce special static invariants and regard them as a possible future addition.



		non-private method	private method	constructor
invariant	entry	yes	no	no
	regular exit	yes	no	yes
	exception	yes	no	no
precondition	entry	yes	yes	yes
	regular exit	no	no	no
	exception	no	no	no
postcondition	entry	no	no	no
	regular exit	yes	yes	yes
	exception	no	no	no

Table 1: Rules for invariants, pre- and postcondition checking

## Inheritance

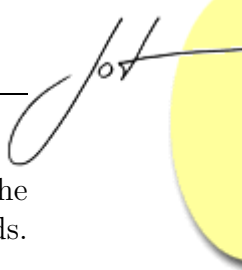
In iContract and JMSAssert, the compiler implicitly refines pre- and postconditions as well as invariants. In Jass, the programmer can use refinement, but he need not do so.

An Eiffel programmer who redefines pre- or postconditions must add the new conditions to the overridden methods' conditions by using the keywords `require else` and `ensure then`. The semantics equal those of `or else` and `and then`. Redefinitions may weaken preconditions and strengthen postconditions. Assertions of overridden methods are evaluated only if necessary. As pointed out by Findler and Felleisen [FF01] combining assertions of supertypes and subtypes in this way may not always reflect the programmer's intention. We regard static checks to detect such problems as important future additions to our work and do not deal with this topic in the present paper.

In our proposal we can associate redefined methods with pre- and postconditions. They will be implicitly concatenated with the pre- and postconditions of the overridden methods using non-commutative versions of *or* and *and*, respectively. As in Eiffel, redefined methods retain the conditions of overridden methods at the absence of `@require` and `@ensure` clauses. In this frequent situation the programmer need not write anything special. Pre- and postcondition checks first ensure that the conditions of the invoked methods are satisfied. On failure (for preconditions) or success (for postconditions) the conditions of the overridden methods have to be evaluated. Pre- and postconditions of overridden methods do not get evaluated in other cases.

Invariants are conjuncted across class extensions. Invariant checks ensure that the invariant of the class itself, the invariants of all base classes, and the invariants of all implemented interfaces are satisfied. For inner classes the invariants of the enclosing classes (and their base classes and implemented interfaces) have to be satisfied, too. Methods in classes not containing assertions or with disabled assertions do not perform invariant checks. If these methods are not overridden in a subclass,





then these methods do not perform invariant checks in the subclass even if the subclass uses assertions. Static and private methods never override other methods. Therefore, they cannot inherit constraints.

## Old Values and Results

Postconditions can refer to the old object state (this is the state at the time of method invocation) by using the special syntax `@@(expr)` and to the method's return value by `@@()`. For example,

```
@ensure { assert (@@(deposit) > deposit, "no deposit increase"); }
```

ensures that the value of the variable `deposit` was increased while executing the method body. A more intuitive name like `old` instead of `@@` cannot be used because of legacy code in which `old` occurs as identifier.

To make `@@` expressions more powerful, the argument of `@@` can contain all kinds of expressions, not just field accesses. Such expressions get evaluated immediately after method invocation. In the example in Fig. 6, the first assertion ensures just the identity of the old value of `vector` with that of the current value. The second assertion compares the vectors for equality; however, only the current state of the old value of `vector` is compared with the current state of the current value. To compare the old state of the old value with the current state of the current value we need the third assertion. The last assertion shows a comparison of a selected field; in this case only a single value of type `int` is saved until the method execution terminates.

```
private Vector vector;  
...  
assert @@(vector) == vector;  
assert @@(vector).equals(vector);  
assert @@(vector.clone()).equals(vector);  
assert @@(vector.size()) < vector.size();
```

Figure 6: Examples for `@@` expressions

An `@@` expression is evaluated immediately after invariant and precondition checking, and the value is stored in an inner object until it is used in the postcondition. Hence, the side effects of `@@` expressions are visible within the method body although the expressions belong to the postcondition. Cascading `@@` makes no sense and causes compile-time errors.

Field declarations in the body of an interface are implicitly static and final. Therefore, postconditions on old values of fields are meaningless and not supported. Of course, the result value is accessible.

In contrast to preconditions of methods, constructors cannot access object fields in preconditions and use `@(expr)` and `@()` in postconditions because the fields are not yet initialized. Although the base class constructor invocation in a constructor must be the first statement in the Java source code, the constructor's precondition has to be checked before. Source-to-source translators cannot perform these checks correctly.

## 4 TRANSLATION


The Kopi project is a Java software project of Decision Management Systems (DMS), a small Austrian/American/French software company. It provides a framework for developing database applications using Java, JDBC and SWING. Kopi contains a set of tools like DIS (Java disassembler), KSM (Java assembler) and KJC (Kopi Java Compiler). KJC compiles Java source code to JVM byte code. It is a Java compiler written entirely in Java and available under the terms of the GNU Public License. We extended KJC with support for genericity and Design by Contract as described in Sect. 3 and [Lac01]. This compiler is available from DMS (<http://www.dms.at/kopi/>).

Subsequently we describe how KJC translates preconditions, postconditions, and invariants to JVM code. In a nutshell, the translation generates separate methods for class invariants and `@require` and `@ensure` clauses. To minimize the possibility of name clashes we use the `$` character in every compiler-generated identifier; as defined in the Java language specification §3.8 [GJSB98], this character shall be used only in mechanically generated code.

### Translation of Invariants and Preconditions

We translate the invariant of a class to a method `protected void $invariant()`. Each non-static exported method in the class invokes this method once at the entry and once at the end, and each constructor does so only at the end. The `assert` statements are translated to `if` statements that throw `InvariantError`.

The precondition of a method `foo` is compiled into a separate method `foo$pre` of return type `void`. Modifiers of pre- and postcondition methods are the same as those of the original methods except for `abstract` and access modifiers. Pre- and postcondition methods cannot be abstract. They are private if the original methods are private; otherwise they are protected. The formal parameters correspond to those of `foo`, but they are all `final` to forbid changes of their values. There is an extra `final` parameter of name `$class` with the class containing the method as parameter type if the method is neither `private` nor `static`. This parameter avoids overriding with pre- and postcondition methods of the same names in subclasses, but causes these methods to be overloaded (because of different parameter types). The `$class` parameter is also needed for assertion checks in interfaces (see below).



```

protected void $invariant() { if (!(getDeposit() > 0)) ... }
protected void getDeposit$pre (final Account $class) { ... }
public int getDeposit() {
    if (!Account.$assertionsDisabled) // enabled/disabled assertions
        if (!AssertionRuntime.tstandsetRunAssertion()) {
            try {
                this.$invariant(); // check invariant
                this.getDeposit$pre(null); // check precondition
            } finally {
                AssertionRuntime.clrRunAssertion();
            }
        }
    try {
        ... // method body
    } finally {
        if (!Account.$assertionsDisabled) // enabled/disabled assertions
            if (!AssertionRuntime.tstandsetRunAssertion()) {
                try {
                    this.$invariant(); // check invariant
                } finally {
                    AssertionRuntime.clrRunAssertion();
                }
            }
    }
}
}

```

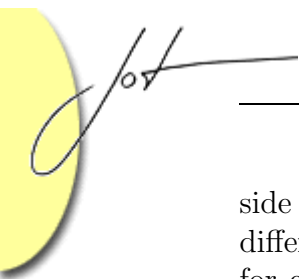
Figure 7: Invariant and precondition checking in a method of class `Account`

As simple assertions in Java, the invariant and all pre- and postconditions in a class can be disabled. The class loader sets the value of the static, private variable `$assertionsDisabled` in each class to `true` if assertions are disabled, otherwise to `false`. The example in Fig. 7 (see Fig. 5 for the source code) shows that all kinds of assertions are checked only if this variable contains `false`.

Further assertion checks shall be switched off while checking an assertion. Otherwise invariant checking would result in infinite recursion (see Fig. 7). To prevent assertion checking within assertions we store in the special class `AssertionRuntime` the sort of code (regular code or assertions) currently executed by each thread. This simple database is consulted and updated before each assertion check.<sup>1</sup>

Preconditions and invariants are checked only if `$assertionsDisabled` is `false` and `tstandsetRunAssertion` returns `false` for the current thread (see Fig. 7). As a

<sup>1</sup>It would be simpler and more efficient to add a corresponding flag to `Thread`. However, we avoided that solution because `Thread` is part of the standard Java environment that we do not want to change.



side effect `tstandsetRunAssertion` sets a flag for the current running thread. Since different threads are allowed to execute assertions at the same time, a separate flag for each thread has to be reserved. After the assertions have been evaluated this flag has to be cleared by `clrRunAssertion`. The last clearing of the flag occurs in a `finally` block to guarantee the clearing even if exceptions were thrown.

## Translation of Postconditions and @@ Expressions

References to old values and result values in postconditions make the translation more complex. To avoid name clashes the method created for the postcondition of a method `foo` gets name `foo$post` if `foo` returns a value and `foo$V$post` if the return type is `void`. Postcondition methods always have the return type `void`. Their formal parameter lists include a parameter `$storage` of the type of an inner class that stores old values if `@@(expr)` expressions occur in the preconditions or the preconditions of corresponding overridden methods. For non-private, non-static methods there are `$class` parameters as in precondition methods. For methods returning values there is a further parameter `$result` standing for the returned value. We translate `@@()` expressions to accesses of `$result`. All parameters are declared `final`.

It is not feasible to store old values in local variables because old values may be needed even for methods overridden in a subclass; overriding methods have no access to private variables. Hence we must use inner classes. The compiler generates names of inner classes to store old values by adding successive numbers to `$$Store`. Each `@@(expr)` expression introduces a new `final` variable of name `$field` (with an appended successive number) in the class. The variable gets initialized with `expr`.

Fig. 8 shows an example of a postcondition method and an inner class storing old values. The postcondition method cannot be invoked in the `finally` block because it shall not be executed on exceptional termination. The right place for the postcondition check is the return statement.

## Inheritance and Interface Implementations

In the presence of an overridden method a precondition check shall fail only if the precondition of the invoked method fails and then the precondition of the overridden method fails, too. Fig. 9 shows an example of a precondition method for an overriding method. A failing assertion causes the precondition of the overridden method to be evaluated. If this precondition also fails, the first generated exception is updated and thrown again.

As shown in Fig. 9, the translation of inheritance for postconditions is slightly simpler. If any check of (partial) postconditions for overridden methods fails, the whole postcondition check fails.

Methods inherit the `Store$$` classes from overridden methods. If they need their



```
protected class $$Store0 {
    final int field$0 = Account.this.value;
}
protected void withdraw$post (final $$Store0 $storage,
    final Account $class, final int $return, final int withdrawal) {
    if (!($return > this.max_overdrawn))
        throw new PostconditionError("Account.java: 22");
    if (!($storage.field$0 < this.value))
        throw new PostconditionError("Account.java: 23");
}
public int withdraw (int withdrawal) {
    $$Store0 $storage = null;
    if ... { // if assertions shall be checked
        ... // set mode; invariant and precondition checks
        $storage = new $$Store0(); // store old values
        ... // reset mode
    }
    try {
        ... // method body
        int $return = ...; // set value to be returned
        if ... { // if assertions shall be checked
            ... // set mode
            this.withdraw$post($storage, null, $return, withdrawal);
            ... // reset mode
        }
        return $return;
    } finally {
        ... // invariant check if necessary
    }
}
```

Figure 8: Example of a store class and postcondition method

```

protected void withdraw$pre (final Account $class,
                             final int withdrawal) {
    try {
        ... // require clause of current method
    } catch (PreconditionError pe1) {
        try {
            super.deposit$pre(deposit);
        } catch (PreconditionError pe2) {
            pe1.setSuperPreconditionError(pe2);
            throw pe1;
        }
    }
}

protected void withdraw$post (final Store$$0 $storage,
                              final Account $class,
                              final int $result,
                              final int withdrawal) {
    ... // ensure clause of current method
    super.withdraw$post ($storage, null, $result, withdrawal);
}

```

Figure 9: Translation of a precondition and postcondition with inheritance

own such classes for old values, then the own class must extend the inherited one. The interface of the postcondition methods in a class depends on the postcondition methods in the base class. This dependence adds some complexity to the compiler development because base classes must be translated before their subclasses.

With generic classes as in GJ [GW98] and in KJC [Lac01] it can happen that a method overrides (indirectly) two or more methods. This property requires the translation to be a little bit more sophisticated. Since Java does not (yet) support genericity we do not deal with these topics in this paper.

If a class extends another class or implements interfaces, the generated invariant method invokes the corresponding methods of the base class or interfaces as shown in Fig. 10. We put methods generated for preconditions, postconditions and invariants in an interface **X** into a new class of name **X\$\$\$Assertions**. The idea of introducing classes with special names to store information about interfaces is used, for example, in standard Java for beans [Ham97]. All methods in this class are static. We use the parameter **\$class** to refer to the corresponding instance of the interface. The argument provided for this parameter must be different from **null**.

In the same way, pre- and postcondition methods must not only call the corresponding methods of overridden methods, but also those of methods in interfaces implemented by the class. Old values cannot occur in postconditions of methods in interfaces. Hence, such postcondition methods need no parameter **\$storage**.



```
public class A extends B implements X, Y {
    protected void $invariant() {
        super.$invariant();
        X$$$Assertions.$invariant(this);
        Y$$$Assertions.$invariant(this);
        ...
    }
}
```

Figure 10: Translation of invariants

## Translation of Constructors

In essence the translation of pre- and postconditions in constructors resembles that for ordinary methods. Major differences include the fact that invariants have to be checked for private constructors, and invariants shall not be checked on exceptional termination. Hence, invariant checks resemble postcondition checks in ordinary methods without return values. Assertion checking is even simpler for constructors because constructors in subclasses do not override constructors in base classes; they just overload them.

As already mentioned, precondition checks shall be performed before invoking constructors of base classes. In standard Java code the invocation of a base class constructor must be the first action within a constructor. This restriction prevents proper precondition checks. Fortunately, JVM byte code does not enforce this Java rule so that a Java compiler can construct correct code for precondition checks.

In classes where no explicit constructor is defined, the Java compiler introduces an empty default constructor without parameters. Default constructors must check the invariant as each other constructor does.

## 5 PERFORMANCE RESULTS

To evaluate the overhead imposed by assertion checks we have completed several classes with assertions. All measurements were performed on a Celeron 533 with SUN JDK Version 1.4.0 Beta 2 running WindowsNT 4.0. Timing result values show the averages of the times measured for repeated test runs without the fastest and the slowest ten percent.

We have chosen a small application and some micro benchmarks which heavily use pre- and postconditions and separately invariants. The first program evaluated was the compiler-interpreter for Mini (a small subset of the Java language) written in Java. The compiler generates code for a modified small subset of the Java Virtual Machine. Every method of the lexical analyzer and the code generator contains assertions. The whole program contains 21 assertions.

Mini program length [byte]	original code [ms]	constrained code		empty constraints enabled [ms]
		disabled [ms]	enabled [ms]	
588	75	115 (+53%)	170 (+126%)	130 (+73%)
833	90	130 (+44%)	200 (+122%)	145 (+61%)
1470	110	150 (+36%)	270 (+145%)	190 (+72%)

Table 2: Mini compiler-interpreter without printing

We run the compiler with three different input data sets. The first column in the table 2 shows the sizes of the three different Mini programs. The second column gives run times for the original code without any assertions. In the next two columns we present run times for the code with assertions, one column with disabled assertions and the other one with enabled assertions. For binary compatibility every method has a pre- and a postcondition method unless the pre- or postcondition is empty. To separate the cost of the assertions from the cost of the assertion managing code, the last column shows run times of the code containing preconditions, postconditions and invariants, but these assertions are empty.

It can be seen that the overhead between the original program and the program with disabled assertions is always about 40 ms, independent of the program length. The relative overhead of assertion checking decreases with bigger input data.

The Mini compiler-interpreter has only static methods. So the results in table 2 reflect the overhead for pre- and postcondition checking, but not for class invariant checking.

Next, the impact of assertions to the performance of a binary search tree which is used for minimum and maximum search is evaluated. The performance depends on the distribution of the elements. We use three different data sets. The first one guarantees trees to be balanced, where the differences between the depths of the left and right branches must not be larger than one. The second set uses a random generator, and the last one inserts elements in ascending order. The postcondition of the insert method guarantees that (1) if the minimum of the tree is greater than the new element, then the new minimum returned by `getMin()` equals the new element, and (2) if the maximum is smaller than the new element, then the new maximum return by `getMax()` is equal to the new value.

insert strategy	original [ms]	constrained code		empty constr. enabled [ms]
		disabled [ms]	enabled [ms]	
perfect	26	26 (+ 0%)	200 (+ 669%)	93 (+ 257%)
random	30	30 (+ 0%)	250 (+ 733%)	120 (+ 300%)
ascending	90	110 (+22%)	125020 (+138811%)	2249 (+2398%)

Table 3: Binary search tree



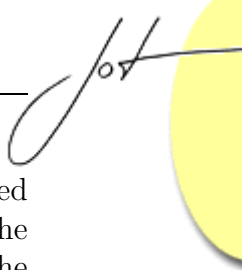


Table 3 shows that if the elements are inserted in ascending order, the constructed tree is in fact a linear list where the element representing the minimum is at the root and the maximum at the end. The method `getMax()` is called twice in the postcondition, once to get the maximum of the old tree and once to get the maximum of the tree after inserting the new element. This explains the huge overhead.

The evaluation of a balanced binary tree is the content of the next test (see table 4). The invariant ensures that the tree always is balanced.

elements inserted	original code [ms]	constrained code		empty constraints enabled [ms]
		disabled [ms]	enabled [ms]	
500	136	136 (+0%)	460 (+238%)	254 (+87%)
1000	476	476 (+0%)	1544 (+224%)	749 (+57%)
2000	2217	2221 (+1%)	6232 (+181%)	2827 (+28%)

Table 4: Balanced binary tree

Since the invariant is checked twice, this overhead can be expected.

For the next test assertions were added to the implementation of a stack class from GNU classpath. Every used method of the stack contains one precondition and two postconditions. Elements were added to the stack and then removed again until the stack was empty.

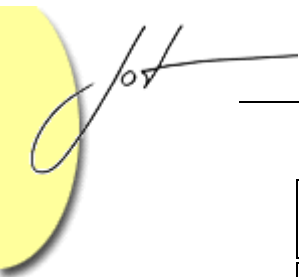
elements inserted	original code [ms]	constrained code		empty constraints enabled [ms]
		disabled [ms]	enabled [ms]	
50000	60	60 (+ 0%)	1517 (+2428%)	1342 (+2136%)
100000	137	152 (+12%)	3046 (+2140%)	2688 (+1876%)
500000	580	654 (+13%)	14988 (+2484%)	13200 (+2176%)

Table 5: Stack

The methods in the stack class are very small. The results in table 5 show that the overhead of assertion checks is quite large. Especially the results for empty preconditions, postconditions and invariants in the last column show that book keeping for assertions takes much more time than the evaluation of the assertions themselves. A major reason for the overhead is frequently repeated tests whether the current execution of a method belongs to an assertion check or not, as explained in Sect. 4.

These tests are performed by methods of the class `AssertionRuntime`. To verify their overhead table 6 shows results for the stack example with empty implementations of the methods of `AssertionsRuntime`.

The complex implementation of `AssertionRuntime` could be replaced with a boolean flag changing the class `java.lang.Thread`. Table 6 makes obvious that



elements inserted	original code [ms]	constrained code		empty constraints enabled [ms]
		disabled [ms]	enabled [ms]	
50000	50	56 (+12%)	130 (+160%)	73 (+46%)
100000	112	127 (+14%)	254 (+126%)	160 (+42%)
500000	578	654 (+13%)	1254 (+116%)	819 (+42%)

Table 6: Stack without assertion management code

such a change of the standard would be very useful to get good performance. Table 7 compares the run times of our unmodified KJC system with that of our system with a modified class `java.lang.Thread`. The table also shows results for some of the other systems discussed in Sect. 2.

elements inserted	measured times with constraints enabled [ms]				
	unmodified KJC	modified KJC	jContractor	iContract	Jass
5000	— <sup>3)</sup>	— <sup>3)</sup>	19192	82	756
10000	300	71	— <sup>1)</sup>	108	2990
50000	1517	419	— <sup>1)</sup>	534	225346
100000	3046	825	— <sup>1)</sup>	1076	— <sup>2)</sup>
500000	14988	4109	— <sup>1)</sup>	5428	— <sup>2)</sup>

<sup>1)</sup> throws the exception `java.lang.OutOfMemory`

<sup>2)</sup> not measured (too long because Jass uses cloning)

<sup>3)</sup> not measured

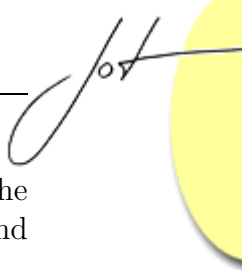
Table 7: Comparison of systems using stack

The difference between a program without assertions and a program with disabled assertions is negligible except if the program is very small.

## 6 FURTHER WORK AND CONCLUSIONS

We added support for Design by Contract to Java. The syntax of our language extension is upward compatible to the Java language specification. Our approach has been implemented as part of the Kopi Java compiler. All class libraries remain unchanged, each class can be compiled separately, and the generated byte code can be executed on any JVM compatible with JDK 1.4. A performance evaluation shows the good performance of our implementation compared to others.

Our approach allows the use of private variables in assertions. Implementing assertions in a compiler gives the advantage of full debugging support and correct positioning of the assertion code which is impossible for precompilers. Our compiler is available at (<http://www.dms.at/kopi/>).



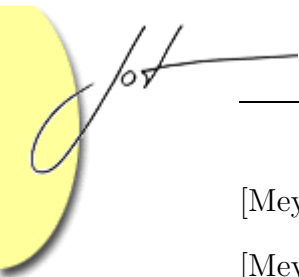
Future work may be the introduction of invariants for static methods. The performance can be improved if JVM implementations are aware of assertions and do further optimizations to remove redundant evaluations.

## ACKNOWLEDGEMENT

We express our thanks to David Gregg for his comments on earlier drafts of this paper. We would also like to thank the reviewers for their helpful suggestions.

## REFERENCES

- [Ber99] Detlef Bertetzko. Parallelität und Vererbung beim "Programmieren mit Vertrag" — Weiterentwicklung von JaWa. Master's thesis, Universität Oldenburg, 1999.
- [DH98] Andrew Duncan and Urs Hölzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, December 9, 1998.
- [FF01] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, pages 1–15, October 2001.
- [GJSB98] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 1998.
- [Gos94] James Gosling, editor. *Oak Language Specification*, 1994.
- [GW98] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. GJ specification. draft paper, May 1998.
- [Ham97] Graham Hamilton. JavaBeans. Technical report, Sun Microsystems, July 1997.
- [KHB98] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A Reflective Java Library to Support Design By Contract. Technical report, Department of Computer Science, University of California, December 1998.
- [Kra98] Reto Kramer. iContract — the Java Designs by Contract tool. In *Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26, Santa Barbara/CA, USA*. IEEE Press, Los Alamitos, 1998.
- [Lac01] Martin Lackner. Extending Java. Master's thesis, TU Wien, May 2001.



- [Mey91] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.
- [Mey92] Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, October 1992.
- [Mey98] Bertrand Meyer. *Object Oriented Software Construction*. Prentice-Hall, 1998.
- [Pla00] Michael Plath. Trace-Zusicherungen in Jass — Erweiterung des Konzepts ”Programmieren mit Vertrag”. Master’s thesis, Universität Oldenburg, 2000.
- [Ran00] K. Rangarajan. Design by Contract for Java using JMSAssert, 2000.
- [WK99] Jos Warmer and Anneke Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, 12(1):10–13,28, March 1999.

**Martin Lackner** did his Master’s thesis in 2001 at Technische Universität Wien adding genericity and support for Design by Contract to the Kopi Java compiler. He now works at DMS on the Kopi Java compiler.

**Andreas Krall** is an associate professor at Technische Universität Wien. He has developed the CACAO JVM and is the manager of the Christian Doppler laboratory *Compilation Techniques for Embedded Processors*. He received his Ph.D. in 1988 from the Technische Universität Wien. Send requests regarding this article to [andi@complang.tuwien.ac.at](mailto:andi@complang.tuwien.ac.at).

**Franz Puntigam** is an associate professor at Technische Universität Wien. His main research focus is object oriented languages and type systems for concurrent languages. He received his Ph.D. in 1992 from the Technische Universität Wien.