©2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. This is the author's version. The original publication is available at: http://dx.doi.org/10.1109/ICSE.2007.60

Overview and Evaluation of Constraint Validation Approaches in Java

Lorenz Froihofer, Gerhard Glos, Johannes Osrael, and Karl M. Goeschka
Vienna University of Technology
Institute of Information Systems
Argentinierstrasse 8/184-1
1040 Vienna, Austria
{*lorenz.froihofer|johannes.osrael|karl.goeschka*}*@tuwien.ac.at*

Abstract

Integrity is a dependability attribute partially ensured through runtime validation of integrity constraints. A wide range of different constraint validation approaches exists—ranging from simple `if` *conditions over explicit constraint validation methods and contract specifications to constraints as first class runtime entities of an application. However, increased support for explicitness and flexibility often comes at the price of increased performance costs. To address this issue, we contribute with an overview and evaluation of different constraint validation approaches for the Java programming language with respect to implementation, maintainability and performance. Our results show that the benefits of some of the more advanced approaches are certainly worth their costs by introducing a runtime overhead of only two to ten times the runtime of the fastest approach while other approaches introduce runtime overheads of more than 100, which might be simply too slow in certain applications.*

1. Introduction

Constraint validation is one of the most essential tasks of a system to ensure integrity—an important attribute of dependability and security [1]. The integrity constraints are defined according to an application's requirements and explicitly stated (and probably negotiated with stakeholders) during the requirements analysis phase. Consequently, they represent a subset of an application's requirements that should be ensured by the implementation. Being typically stated only informally in the requirements analysis phase, e.g., written down in natural language, the constraints are often more formally described and attached to the application model in the design phase. The Unified Modeling Language (UML), for example, provides the Object Constraint Language (OCL) to explicitly specify constraints—in addition to the possibility to already express some constraints, e.g., cardinality or XOR, in the graphical notation of UML.

In contrast, the constraints are most often no longer explicitly available in a system's implementation, i.e., the constraint validation code is often tangled with code for the business logic. For some applications, this might be a reasonable or satisfying approach. However, just following the simple approach to use `if` statements to validate constraints, for example, often turns out not to be the best solution due to several reasons: (i) A single constraint might have to be checked in different places in the program, which might lead to inconsistent implementations of the same constraint throughout the implementation code. This also makes it difficult to verify that the constraints specified during system analysis and design have actually found their way into the implementation. (ii) Furthermore, constraints might also describe contracts between different system entities. Unfortunately, an implicit constraint implementation does not support this design-by-contract principle [13]. (iii) Some systems might even be more demanding by requiring explicit runtime handling of integrity constraints, e.g., to support object versioning [6] or adaptive dependability by (temporarily) relaxing integrity requirements [5].

Several constraint validation techniques for the Java programming language have been developed in the past. Most of these techniques are inspired by the way constraints are integrated in the Eiffel programming language—by building upon the design-by-contract principle. While this principle has a strong focus on the detection of contract violations between producers and users of a certain piece of code, e.g, between the writer and the caller of a method, other approaches focus on how system integrity can be achieved via validation of integrity constraints. Besides this slightly different focus, both approaches aim at improved system dependability through runtime constraint validation. Each of these constraint validation techniques has its advantages

and disadvantages. Increasing the explicitness and flexibility of constraint handling and enforcement generally introduces runtime overheads. Motivated by these considerations and our requirement for explicit runtime handling of constraints to support adaptive dependability [5], we contribute with an overview and evaluation of different constraint validation approaches in Java.

Paper overview. Section 2 provides an overview of different constraint validation approaches in Java. In Section 3 we discuss some advantages and disadvantages of the approaches with respect to implementation and maintainability. Section 4 provides a performance evaluation and Section 5 provides an overview of related work. Finally, we conclude our paper and identify future research challenges in Section 6.

2. Constraint implementation

Integrity constraints are primarily stated in the way of pre- and postconditions and invariant constraints [13]. Preconditions are bound to methods and have to be satisfied before a method is invoked. Postconditions are also bound to methods and have to be satisfied when the method returns. Invariant constraints are bound to the context of a class. However, the point in time of when to check (*trigger point*) invariant constraints is not standardized. Different trigger points are possible, e.g., check before and/or after the invocation of public or all methods of the class for which the invariant is defined. However, an invariant constraint has to be checked at least after a call to a method that may lead to a violation of the constraint. To unify this issue and make our performance evaluations comparable, we check invariant constraints before and after the invocation of public methods of our reference application—complying with design-by-contract in the way that if and only if an invariant holds before a public method invocation, it must also hold after the method invocation.

This section provides an introduction to different constraint validation approaches for Java. We start from the simple approach of handcrafted constraints and continue to more flexible approaches such as constraint code generation and explicit runtime constraints. While this section will shortly describe how each approach is performed, sections 3 and 4 will discuss their advantages and disadvantages with respect to implementation, maintainability, and performance.

2.1. Handcrafted constraints

The most simple approach to implement constraint checking in Java is to tangle the constraint checking code with other code, e.g., for the business logic. This approach does not require any formal specification of constraints as the transformation from the integrity requirement to the constraint implementation has to be performed by the programmer. Generally, the result is one or more `if`-statements to check a certain condition and act according to whether these statements are `true` or `false`. A source code example for this approach is provided in Listing 1.

Listing 1. Simple constraint implementation

```
class A {
  void someMethod() {
    if ( ( (...) OR (...) ) AND (...) ) {
      //business logic code
    } else if (! ...) {
      throw SomeException();
    } else {
      if (...) {
        //exception handling code
        printErrorMessage(...);
      } else {
        //business logic continued
      }
    }
    //further business logic
  }
}
```

2.2. Code instrumentation

Code instrumentation refers to injecting automatically generated code into a piece of original code, e.g., to add some additional functionality to the existing code. With respect to Java, we differentiate between source code and byte code instrumentation, depending on whether the Java sources are instrumented with Java code before compilation (i.e., pre-compiler approaches) or byte code is instrumented with byte code after the Java sources are already compiled. Within this section, we focus on source code instrumentation and will discuss differences to byte code instrumentation at the end of Section 3.3.

However, common to both approaches is the requirement that the injected code is generated according to constraints specified in a constraint language known by a certain tool. Generally, tools use either UML class models paired with OCL constraints [19, 21], constraints defined within Java comments, or methods with names according to naming conventions. The languages for constraint specification in the latter cases range from Java code [8] over OCL and OCL-like expressions [10] to tool-specific constraint languages [12]. For source code instrumentation, the two primary approaches are in-place code injection and wrapper-based validation:

In-place validation code. This approach injects code for constraint checks directly at the place where the validation should be performed, e.g., within the performed method. If a constraint affects several methods of possibly different

objects, constraint checking code for the same constraint is injected at any place where constraint checking is required. This approach is illustrated in Listing 2. An example for this approach is the iContract tool [10].

Listing 2. In-place code injection

```
public int countChar (char c) {
  //--> code for validation of invariant
  //     constraints and preconditions
  //BEGIN original code
  int result = 0;
  char[] chars = toCharArray();
  for (int i=0; i<chars.length; i++) {
    if (chars[i] == c) result++;
  }
  //END original code
  //--> code for validation of invariant
  //     constraints and postconditions
  return result;
}
```

Wrapper based constraint validation. In this case, methods restricted by constraints are wrapped and the constraint validation code is contained in the wrapper method. Generally, the original method is renamed and only called via the wrapper method. For example, the original method `countChar` would be renamed to `countChar_wrapped` and the wrapper method would be named `countChar`, see Listing 3. Consequently, calls to `countChar` execute the wrapper method including the constraint validation code. A typical example for this approach is the Dresden OCL toolkit [21].

Listing 3. Wrapper-based validation

```
public int countChar (char c) {
  //--> code for validation of invariant
  //     constraints and preconditions
  //--> Call the original method
  int result = countChar_wrapped(c);
  //--> code for validation of invariant
  //     constraints and postconditions
  return result;
}

public int countChar_wrapped(char c) {
  int result = 0;
  char[] chars = toCharArray();
  for (int i=0; i<length(); i++) {
    if (chars[i] == c) result++;
  }
  return result;
}
```

2.3. Compiler-based approaches

Compiler-based approaches build upon a specific Java compiler that is enhanced with functionality to read constraint specifications and integrate the corresponding constraint validation mechanisms into the Java byte code. In contrast to code instrumentation approaches, the transformation from source code to constraint checking byte-code is performed in a single step. An example for a compiler-based approach is the Java Modeling Language (JML) [12]. Listing 4 provides an example for an input to a compiler-based approach that does not use a custom extension of the Java programming language.

Listing 4. Compiler-based constraint checks

```
/**
 * @pre o != null;
 * @post size() == size()@pre + 1;
 */
public void add(Object o) {...}
```

2.4. Explicit constraint classes.

Encoding constraint validation code in explicit Java classes is an approach that completely separates validation code from, e.g., the code for the business logic. For example, the constraint validation code may be contained in a `validate` method that is executed with appropriate arguments whenever a certain constraint has to be checked. This approach requires appropriate trigger mechanisms to ensure that the `validate` method is called whenever necessary. Trigger mechanisms include explicit code statements made by the programmer, in-place code generation of the calls, wrapper-based approaches, and interceptor mechanisms discussed in Section 2.5. Explicit constraint classes are used, for example, in [5, 19]. Listing 5 illustrates this approach.

Listing 5. Explicit constraint classes

```
public class ExampleConstraint {
  public boolean validate(Object o) {
    boolean result = false;
    //--> Check the constraint and set the
    //     return value ''result'' depending on
    //     whether the constraint is satisfied.
    return result;
  }
}
```

Constraint repository. Encapsulating the constraint checking code in separate classes allows for more flexible handling of integrity constraints. For example, all constraints of an application can be registered within a constraint repository. At any point in time, this repository can be queried for constraints based on different criteria such as the class of the invoked object or the signature of invoked methods. Consequently, preconditions, postconditions, and invariant constraints affected by method invocations can be queried from the constraint repository. Moreover, using such a constraint repository allows to add, remove, enable, and disable integrity constraints even during runtime.

2.5. Interceptor mechanisms

Interceptor mechanisms provide the possibility to intercept different events such as the call to a method. Subsequently, the interceptor can perform some actions and continue or possibly abort the current action, e.g., the method call. Hence, interceptor mechanisms are an appropriate mechanism to implement so called "cross-cutting concerns" such as logging or—in our case—constraint validation. For constraint validation, one can either implement the constraint checking code within the interceptor [20] or use a generic interceptor that redirects calls, e.g., to explicit constraint validation classes (Section 2.4). The second approach can be achieved by combining the interception mechanism with a constraint repository. Within the following paragraphs, we describe the major interceptor mechanisms available for the Java programming language:

Aspect-oriented programming. Aspect-oriented programming (AOP) [9] is closely related to code instrumentation as AOP is often achieved through (byte) code instrumentation. It follows the programming paradigm of interception and weaving of so called aspects into program execution. A typical example for the weaving of an aspect is to introduce logging functionality to an existing program. Today, several tools supporting the AOP programming paradigm already exist (http://aosd.net). Within this paper, we concentrate on AspectJ and JBoss AOP as two well-known tools with a significant user base.

Proxy implementations. Since version 1.3, the Java programming language provides the concept of a proxy implementation for interfaces (`java.lang.reflect.Proxy`). If a method is invoked on the proxy, the registered invocation handler is notified with details about which method was invoked on which object with which arguments.

CORBA and EJB interceptors CORBA and Enterprise JavaBeans (EJB) also provide the possibility to intercept method invocations as both technologies separate interface from implementation. Hence, the interceptor mechanisms of CORBA and EJB can be used as trigger mechanisms for constraint validation. However, within this paper we concentrate on plain Java applications not building upon higher level specifications.

2.6 Summary

Table 1 summarizes this section with an overview of the most influential tools supporting constraint validation. For each tool, we provide how constraints are specified and which mechanism is used for integration of constraint checks.

Table 1. Constraint validation tools

Name and reference	Constraint specification	Integration of constraint checks
Dresden OCL toolkit [21]	OCL constraints defined for a UML class model	Wrapper-based source code instrumentation
Handshake [4]	Custom language in a separate file	Runtime byte code instrumentation on class load time
iContract [10]	OCL in custom tags of Java comments	Source code instrumentation
Jass [2]	Custom language in special Java comments	Source code instrumentation
jContractor [8]	Java methods that follow a defined naming convention	Byte code instrumentation by class loader
JML [12]	JML constraints in Java comments or separate file	Custom compiler
JMSAssert	Custom language in custom tags of Java comments	Pre-processor for standard Java compiler, paired with custom library
Kopi Java compiler [11]	Extension of the Java language with certain keywords	Custom compiler
USE [17]	OCL expressions	Runtime interpretation of OCL constraints

3. Implementation and maintainability issues

This section provides a description of implemented constraint validation approaches and discusses several issues with respect to implementation of constraint checks and maintainability of the resulting code.

3.1. Implemented approaches

For evaluation and comparison of the different constraint validation approaches, we implemented the following variants:

- *No checks*: is an implementation of the application without any constraint checks.
- *Handcrafted constraints*: is the case where the constraint checks are manually integrated into the application according to Section 2.1.
- *Dresden OCL Toolkit*: is a wrapper-based approach with tool-generated constraints.
- *JML*: implements a compiler-based approach with manually specified constraints.

- *AspectJ-Interceptor*: is an AOP approach where the constraint validation code is implemented directly in the AspectJ aspect specifications.
- *AspectJ-Repository*: uses explicit constraint classes and a constraint repository to allow flexible runtime handling of constraints.
- *JBoss-Repository*: implements the same as AspectJ-Repository but uses the JBoss AOP toolkit as interception mechanism.
- *Java-Proxy*: uses the Proxy mechanism of Java as interception mechanism and also makes use of the constraint repository to look up affected constraints.

The selection of the different approaches is primarily motivated by our requirement for explicit runtime constraints, e.g., to balance the dependability properties availability and consistency [5]. Consequently, we evaluate the benefits and costs of different explicit runtime constraint checking approaches as well as other constraint checking approaches with better performance or tool support.

As expected, our performance studies showed a major overhead in the repository-based approaches, caused by searching for affected constraints. Hence, each of the repository-based approaches is also evaluated with an optimized repository that performs caching of query results. In this case, a subsequent query for constraints based on the same criteria reduces to a lookup in a hash table with a key that combines our search criteria: class, method and constraint type.

3.2. Handcrafted constraints

With handcrafted constraint checks, the programmer retains full control over where, when and what is to be checked and how to react on violations. However, the programmer is also responsible for accurately documenting these checks and to update them if the integrity requirements of an application change. Unfortunately, this tends to lead to inconsistencies between application requirements, documentation, and implementation of constraints. Moreover, the same constraint might be implemented differently (and inconsistently) at several places within an application.

3.3. Code instrumentation

The main advantage of code instrumentation approaches is that they allow a separation of business logic code from constraint checking code at design and implementation level. The main disadvantage is that the original code is changed through code injection. Several works [18, 21] already investigated constraint implementation by using code instrumentation and found that code instrumentation approaches generally suffer from different problems we can summarize as follows:

Return statement: The code for checking postconditions and invariant constraints must be executed before the `return` statement. Hence, the result must be available before the method actually returns. This poses problems when the result to be computed is declared within the return statement itself, e.g., `return 2*x`.

Code duplication: Due to control flow issues, there may be several return points for a method. Therefore, it is necessary to insert the code for checking postconditions and invariants at several points in the same method. Moreover, it could be necessary that constraints have to be checked in more than one method. This leads to even more duplication of constraint checking code.

Reachable point: For methods of return type void it has to be decided whether it is sufficient to only insert the code for checking postconditions at the end of the method. This depends on whether the end is a reachable point of code—or more specifically, executed for every method invocation. Hence, complex control flow analysis is required.

Super statement: If a subclass calls the constructor of a superclass (by invoking `super()`) in Java, the compiler allows no other statements in advance. Therefore, special measures must be taken to implement checks for preconditions and class invariants. Moreover, wrapper-based approaches may lead to infinite loops. For example, if a method `m_wrapped()` of a subclass calls `super.m()`, `m()` of the superclass subsequently will—due to polymorphism—call `m_wrapped()` of the subclass again, thereby introducing an infinite loop. Consequently, appropriate measures, such as adding the class name to the name of the wrapped function, have to be taken.

Pollution of application code: The instrumented code is cluttered with constraint checking statements.

Shift of line numbers: Line numbers shown in compiler messages or stack traces of exceptions point to the modified source code instead of the original code. Hence, the mapping of the line numbers from modified code to original code has to be performed manually by the developer.

Black-box instrumentation: The developer loses control over code changes, as they happen in a black box fashion. This may further lead to unexpected behaviour and performance losses during runtime of the program.

Debugging: Debugging becomes more difficult as standard debuggers will only allow the debugging of the tangled instrumented source code. The shift of line numbers issue described above makes debugging even more complex.

Naming conflicts: Conflicts in the names of, for example, methods or variables must be prevented between the original code and the generated code, since the generated code may define variables or helper methods.

Source code vs. byte code instrumentation. The comparison of source code instrumentation and byte code instrumentation shows that the instrumentation method has a major impact on the generated code fragments:

Source code pre-processing: translates the constraint definitions into standard Java source code which is directly inserted into the original source code. This leads to highly tangled code, which is hard to change and maintain without original sources and constraint definitions.

Byte code post-processing: translates the constraint definitions to Java byte code and injects the generated code into the existing byte code after compilation—either before runtime or dynamically during runtime through the usage of a custom class loader. In any case, this preserves the original source code. Hence, the developer does not recognize any changes to the code, but also has no control over the possibly dynamically instrumented code. This leads to difficulties in debugging, possibly unexpected behaviour during runtime, and a performance loss because of the constraint checks and—if performed at runtime—the dynamic code instrumentation through the class loader. However, some issues can be solved by byte code instrumentation compared to source-code instrumentation. For example, on byte code level, the Java Virtual Machine allows arbitrary statements in advance of calling the superclass constructor. This solves the problem of code insertion before a `super()` statement. Another example is that in byte code the computation of variables is separated from control flow statements (branching statements). Branching statements denote the only exit points of methods. Consequently, the issues of inline `return` statements and the determination of insertion points for postconditions are easier to address.

3.4. Compiler-based approaches

Custom compilers are often used if the constraint definitions are not provided within Java comments or Java annotations, i.e., the constraint-enhanced programs use an extended grammar of the standard Java language. In this case, the program code is no longer compilable without the custom compiler—or at least a compiler preprocessor—introducing a dependency on the vendor of the (pre-)compiler. Using a pre-processor falls into the category of code-instrumentation approaches described in Section 3.3 and hence will not be addressed here. Compiler-based approaches perform a direct transformation from source code to Java byte code and integrate the constraint checks during this transformation. As the example of JML shows, compiler-based approaches are also used without extensions to the Java language. While this allows to compile the code with a standard Java compiler, it still does not remove the dependency on the custom compiler for constraint validation.

Generally, the principle that constraint checks are generated out of separate constraint statements is similar to code instrumentation above. Hence, some of the issues described in Section 3.3, e.g., black-box instrumentation, debugging, and naming conflicts, also apply to this approach.

3.5. Constraints encoded in interceptors

Within our studies, we used constraints encoded as aspects in AspectJ as representative for constraints encoded in interceptors. No tool support was available for this approach. Hence, we had to manually code the constraints as AspectJ aspects. This already provides a clear separation between constraint validation code and code for the business logic. While the code for the business logic remains in *.java files, the code for constraint checking is contained in separate *.aj files, defining the constraint checks as AspectJ aspects.

One disadvantage of this approach is the strong coupling of the aspects to the base code. Pointcut definitions specify the interception points for constraint validation. If these definitions exactly match specific method signatures, they are very susceptible to changes in the underlying base code in which case the pointcut definitions have to be changed as well. Refactoring support of Integrated Development Environments (IDEs) could provide support here to improve productivity and reduce errors. General pointcut definitions that match multiple method signatures, e.g., by using wildcards, may on the one hand still be matching after some parts of the original method signature in the underlying code have been changed, but on the other hand may be triggered even when not intended. Such errors are difficult to detect and fix, especially in the context of constraint checking as a failing constraint indicates that the reason is a problem in the base code, rather than a mismatch in the pointcut definitions.

3.6. Explicit constraint classes

Encapsulating the constraint checking code in explicit constraint classes allows for explicit runtime handling of integrity constraints. The degree of flexibility, however, heavily depends on the triggering mechanism for constraint validation. While manual integration or code instrumentation are feasible mechanisms to trigger constraint validation, we focused on the combination of a constraint repository paired with a generic interceptor mechanism. This combination allows for a maximum of flexibility, e.g., to add, remove, enable, or disable integrity constraints during runtime of a system—which would require code modification and recompilation in the other constraint validation approaches. However, this flexibility comes at the price of decreased performance compared to other approaches that manually integrate constraints.

4. Performance studies

Our application scenario for the performance evaluation is the management of projects and employees within a company. Employees participate in projects and perform a certain amount of work on a daily basis. Within this model,

several restrictions apply, e.g., an employee can only handle a certain amount of workload. The application contains a mixture of preconditions, postconditions and invariant constraints—78 constraints in total.

4.1. Comparison conditions

In order to allow for comparison of the different approaches, the validation of integrity constraints was performed in a uniform way. More specifically, we applied the following principles:

Constraint scope: Preconditions, postconditions and invariants only constrain public methods. Public constructors are constrained by invariants, private constructors remain unchecked.

Constraint checking: Constraints are checked before (preconditions) or after (postconditions) the actual code of the guarded method is executed. This also holds true for nested method calls. Invariants are immediately checked after public constructor calls and before and after public methods.

Constraint inheritance: In order to address object substitution and behavioral subtyping, constraints of extended superclasses or implemented interfaces are also taken into account. Preconditions of superclasses and interfaces are concatenated with the logical OR operator. Postconditions and invariants are concatenated with the logical AND operator [3].

Error handling: To exclude runtime differences due to different treatment of constraint violations, the measured application scenario does not violate any integrity constraints. However, in other scenarios we ensured that all the approaches actually check the same number of constraints and also correctly detect constraint violations.

4.2. Results

To measure the performance of the individual approaches, we implemented some use cases within our application scenario and let them run a number of times. To reduce the effects of environmental noise and just-in-time (JIT) compilation, we performed 2500 runs of the same example scenario before we measured another 2500 runs of the example scenario with each constraint validation approach. Each run triggers 4875 checks of invariants, 1097 checks of postconditions, and 433 checks of preconditions. The constraint repository based approaches intercept 1605 methods and trigger 7677 search operations within the repository for each run.

2500 runs of the scenario without constraint checks take 125ms to execute on an AMD Athlon XP 2600+ with 512MB of RAM running under Windows XP. The handcrafted constraints approach is the fastest version, but already runs 35 times slower than the same scenario without constraint checks. However, as this is the fastest approach, it is the baseline for comparison of the other approaches with respect to *additional* overheads. In order to evaluate the additional overheads in detail, we separate the total runtime into slices. As most of these slices do not contain constraint checking code, the application without constraint checks provides the baseline for these comparisons. The overheads are calculated according to Formula 1.

$$Overhead = \frac{Runtime\ for\ approach}{Runtime\ for\ baseline} \quad (1)$$

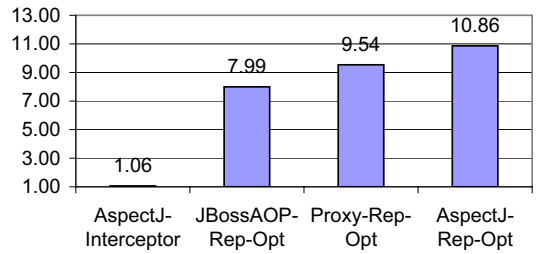

Figure 1. Fastest approaches

Figure 1 provides an overview of the fastest constraint validation approaches where the handcrafted constraints approach provides the baseline. This figure shows that constraints integrated as aspects in AspectJ are almost as fast as handcrafted inline constraints. The overhead introduced is only 1.06 times the runtime of the handcrafted constraints approach. The second fastest approach uses JBoss AOP for invocation interception and an optimized constraint repository containing explicit constraint classes. This introduces a runtime overhead of 7.99. Using a Java proxy with an optimized repository runs 9.54 times slower than handcrafted constraints and AspectJ with an optimized repository shows an overhead factor of 10.86.

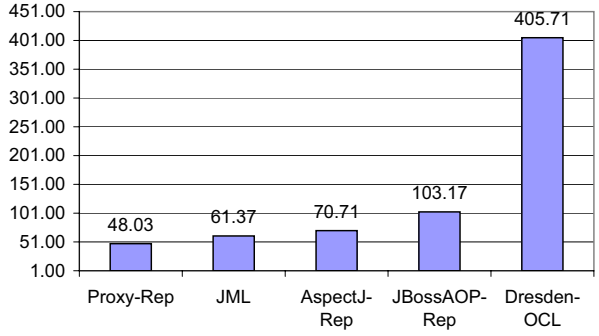

Figure 2. Slowest approaches

Figure 2 provides a comparison of the slowest constraint validation approaches where handcrafted constraint checks again provide the baseline for comparison. The approach to use the Java proxy mechanism and a non-optimized constraint repository requires 48.03 times the runtime of the

handcrafted approach—nearly 4.5 times slower than AspectJ with the optimized constraint repository, which was the slowest approach in Figure 1. After the proxy mechanism follows JML requiring 61.37 times the runtime of handcrafted checks. AspectJ with a non-optimized repository shows an overhead factor of 70.71 and JBoss AOP with a non-optimized repository already runs 103.17 times slower than handcrafted constraint checks. Finally, the Dresden OCL toolkit with tool-generated constraints shows a runtime overhead of 405.71 times the runtime of the handcrafted approach.

Interestingly, the order of the different interceptor mechanisms with respect to performance changes for using an optimized and a non-optimized constraint repository. While JBoss AOP is the fastest mechanism with the optimized repository, followed by the Java proxy and AspectJ, the Java proxy approach is the fastest mechanism for the non-optimized repository, followed by AspectJ and JBoss AOP. This is an unexpected result and our investigations showed that the interceptor mechanisms affect the runtime differently, so that the search overhead within the repository is not a constant factor. However, search overhead is not the only overhead introduced in these cases. In total, we separate the overall runtime into five major time slices with respect to introducing constraint checks by using a repository:

R1 is the net application runtime without constraint checks

R2 is the overhead introduced through invocation interception by the different interceptor mechanisms (Java proxy, AspectJ, and JBoss AOP).

R3 provides the overhead to extract search parameters based on the information that the interceptor mechanism provide. This includes getting invoked method, method arguments and/or class of the invoked object.

R4 is the runtime overhead required for searching constraints within the constraint repository.

R5 is the overhead introduced by the constraint checks themselves

Figures 1 and 2 showed a comparison of all approaches considering the total overhead for constraint checking. Further on, we investigate the different runtime slices of the respective overheads to provide a more in-depth comparison of the constraint repository approaches. For JML and the Dresden OCL toolkit, we only considered the total overhead as the methodology is different and cannot reasonably be fitted to the five runtime slices provided above.

Figure 3 shows the search overhead of the optimized and non-optimized constraint repository compared to the application without constraint checks. These versions include the overheads R2, R3, and R4, but do not check constraints (R5). The difference between the optimized and the

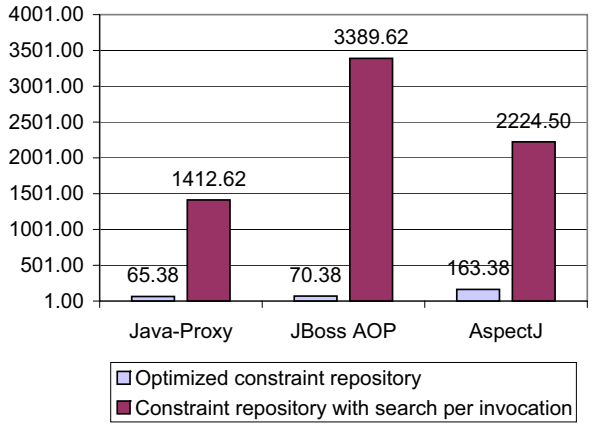

Figure 3. Search overhead (R1+R2+R3+R4)

non-optimized repository is that the runtime overhead R4 is reduced through caching of previous queries. The performance improvements through the optimized constraint repository reduced the overall runtime by a factor between 13.62 (AspectJ) and 48.16 (JBoss AOP). While we configured all of the interceptor mechanisms not to intercept calls performed for searching constraints within the repository, only the Java proxy approach does not modify the Java byte code. Compared to the Java proxy we see an additional runtime overhead introduced by the AOP approaches between 1.07 (JBoss AOP with optimized repository) and 2.50 (AspectJ with optimized repository).

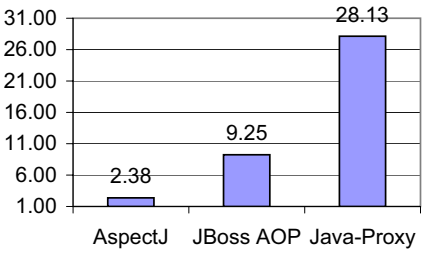

Figure 4. Interception overhead (R1+R2)

Figure 4 illustrates the interception overhead introduced by the different mechanisms. In this case, the intercepted method invocations were immediately forwarded by the interceptors to the called method of the object instance. Hence, the overhead of R1+R2 was compared to R1 (the plain application). This comparison shows that AspectJ provides the fastest interception mechanism, requiring 2.38 times the runtime of the plain application. JBoss AOP shows an overhead factor of 9.25 and the Java proxy requires 28.13 times the runtime of the plain application. As the Java proxy is part of the Java reflection mechanism and does nothing more than invoking the intercepted method via `java.lang.reflect.Method.invoke(...)`, we

primarily attribute this major performance impact to the Java reflection mechanism.

The performance advantage of AspectJ gained through quick interception, however, is lost during parameter extraction. While JBoss AOP and the Java proxy mechanism already provide access to the called Method via a `java.lang.reflect.Method` object, this reference to the method has to be obtained via costly calls to `Object.getClass().getMethod(...)` in AspectJ. Hence, the overhead of R1+R2+R3 compared to R1 provides a different order between the interception mechanisms, ranging from an overhead factor of 19.50 for JBoss AOP over 36.62 for the Java proxy to 98.26 for AspectJ, see Figure 5.

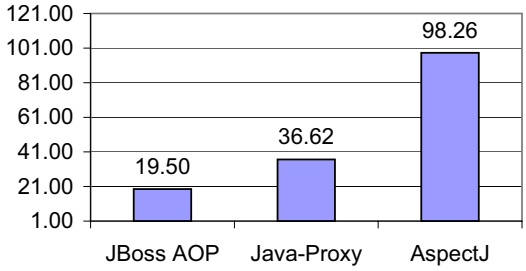

Figure 5. Overhead of invocation interception and parameter extraction for searching the constraint repository (R1+R2+R3)

5. Related Work

Most related work focuses on constraint validation in the sense of design-by-contract [13] as introduced by Meyer for the Eiffel programming language. One focus within this principle is to decide whether the producer or user of a certain piece of code violated the contract. This process is also called the problem of "assigning the blame" for incorrect code. Lackner et al. [11] discuss (pre-)compiler-based approaches supporting design-by-contract in Java. Starting with an overview of Jass [2], iContract [10], jContractor [8], and Handshake [4], they finally describe their own support for design-by-contract in Java through extension of the Java language with new keywords. Furthermore, they provide how this approach was integrated into the Kopi Java compiler and provide performance studies of their approach and a comparison with some of the other approaches. In their studies, the authors experience performance impacts for contract checking code between 2.22 and 1389.11 times the runtime of non-contract checking code. Obviously, these results also have such a wide range of performance impacts as shown by our studies. Plösch [15] provides further details and an evaluation with respect to the degree of assertion support of some tools listed in Table 1. However, these works focus on integrating constraint/contract checks into Java byte code while our evaluation considers wrapper-based source-code instrumentation, byte-code instrumentation as well as interceptor-based approaches to make constraints first class runtime entities.

While design-by-contract might be interesting with respect to distributed software development and contracts between producers and users of code, other works focus on ensuring system integrity in the sense of considering a constraint violation an error that should be treated by corrective measures rather than to view the violation as system failure [1]. Verheecke et al. [19] describe a tool-supported approach to encapsulate constraint checks in explicit constraint classes, generating skeleton code for application classes and constraint checking code in constraint classes based on detailed UML design diagrams annotated with OCL constraints.

There has also been considerable effort to integrate constraint validation mechanisms into object-oriented databases. The approaches are similar to what we described in this paper, ranging from preprocessor-based constraint integration [7] to also considering constraints as first class citizens within an object-oriented database [14].

6. Conclusion and future research challenges

Integrity management in software systems has already been addressed by several researchers and a range of possible solutions exists. However, the selection of an appropriate solution for a specific system will also include implementation, maintainability, and performance considerations. Within this paper we described several constraint validation approaches and contributed by discussing advantages and disadvantages of the different approaches including performance issues.

To sum up, handcrafted constraints showed to be the fastest approach. However, to only separate constraints from, e.g., the code for the business logic, using constraints encoded as aspects in AspectJ is a good choice, requiring only 1.06 times the runtime of handcrafted constraints. If flexible runtime handling of constraints is required, e.g., if it should be possible to add, remove, enable, and/or disable constraints during runtime, an optimized constraint repository paired with the JBoss AOP toolkit as interceptor mechanism should be envisaged. With respect to performance, the automatically generated constraint checks by JML and especially the Dresden OCL toolkit were part of the slower approaches. However, JML provides several tools to thoroughly support the design-by-contract principle. If one is only interested in stating constraints, a thorough support of design-by-contract, and it is acceptable that `java.lang.Errors` are thrown in case of contract violations, JML is certainly a good choice. The Java annotation mechanism introduced in version 5.0 would be an alternative to the definition of constraints/contracts in comments,

allowing also runtime access to the constraints. This approach, however, has not yet been exploited.

Today, we are often thinking in terms of strict consistency and that any threats to integrity—and hence, dependability—have to be avoided and undesirable effects have to be removed or repaired immediately. While this is acceptable for small-scale and tightly-coupled systems, we currently observe a trend towards large-scale integration (systems of systems) and pervasive computing, leading to ultra-large-scale systems [16] in the future. Dependability will be an important aspect of these systems—and integrity management will be part of it. However, strict consistency is not affordable in large- to ultra-large-scale systems. Hence, interesting future research challenges will arise from a transition of thinking in terms of consistency management to thinking in terms of inconsistency management. While this paper focused on "constraints-in-the-small" that are part of software design and programming languages, we will have to think of "constraints-in-the-large" to address the challenges of the future. Such constraints will most probably be fuzzy, imprecise, and potentially require negotiation to decide whether constraints are fulfilled. An analogy that can guide our way in this direction is to view "constraints-in-the-large" as kind of laws, often being precise enough, but sometimes requiring court decisions (negotiations) to decide whether something was actually lawful—or not.

Acknowledgments

This work has been partially funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract 004152, http://www.dedisys.org/).

References

[1] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[2] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. In K. Havel and G. Rosu, editors, *Proceedings of the First Workshop on Runtime Verification*, July 2001.

[3] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 258–267, Washington, DC, USA, 1996. IEEE Computer Society.

[4] A. Duncan and U. Hölzle. Adding contracts to Java with Handshake. Technical Report TRCS98-32, University of California, Santa Barbara, Dec. 1998.

[5] L. Froihofer, J. Osrael, and K. M. Goeschka. Trading integrity for availability by means of explicit runtime constraints. In *Proc. of the 30th Intl. Conf. on Computer Software and Applications*, 2006.

[6] J. S. Goonetillake, T. W. Carnduff, and W. A. Gray. An integrity constraint management framework in engineering design. *Comput. Ind.*, 48(1):29–44, 2002.

[7] H. V. Jagadish and X. Qian. Integrity maintenance in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 469–480. Morgan Kaufmann Publishers Inc., 1992.

[8] M. Karaorman, U. Hölzle, and J. L. Bruno. jContractor: A reflective java library to support design by contract. In P. Cointe, editor, *Reflection*, volume 1616 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 1999.

[9] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[10] R. Kramer. iContract - The Java design by contract tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[11] M. Lackner, A. Krall, and F. Puntigam. Supporting design by contract in java. *Journal of Object Technology*, 1(3):57–76, 2002. Special issue: TOOLS.

[12] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publishers, 1999.

[13] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[14] H. Oakasha, S. Conrad, and G. Saake. Consistency management in object-oriented databases. *Concurrency and Computation: Practice and Experience*, 13(11):955–985, 2001.

[15] R. Plösch. Evaluation of assertion support for the java programming language. *Journal of Object Technology*, 1(3):5–17, 2002.

[16] B. Pollak, editor. *Ultra-Large-Scale Systems*. Software Engineering Institute, Carnegie Mellon University, July 2006.

[17] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 265–277. Springer, 2000.

[18] B. Verheecke. From declarative constraints in conceptual models to explicit constraint classes in implementation models. Master's thesis, Vrije Universiteit Brussel, 2001.

[19] B. Verheecke and R. V. D. Straeten. Specifying and implementing the operational use of constraints in object-oriented applications. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 23–32. Australian Computer Society, Inc., 2002.

[20] Q. Wang and A. Mathur. Interceptor based constraint violation detection. In *ECBS '05: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 457–464, Washington, DC, USA, 2005. IEEE Computer Society.

[21] R. Wiebicke. Utility support for checking OCL business rules in java programs. Master's thesis, Dresden University of Technology, Dec. 2000.