Runtime Constraint Checking Approaches for OCL, A Critical Comparison

Carmen Avila, Amritam Sarcar, Yoonsik Cheon, and Cesar Yeep

TR #10-04
February 2010; revised May 2010

Keywords: design constraints, runtime checking, class invariants, pre and postconditions, aspect-oriented programming, Object Constraints Language (OCL), AspectJ language, JML language.

1998 CR Categories: D.2.2 [*Software Engineering*] Design Tools and Techniques — Modules and interfaces, object-oriented design methods; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

A short version of this report will appear in *SEKE 2010: 22nd International Conference on Software Engineering and Knowledge Engineering, July 1-3, 2010, San Francisco, CA.*

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Runtime Constraint Checking Approaches for OCL, A Critical Comparison

Carmen Avila
Dept. of Computer Science
University of Texas at El Paso
El Paso, TX 79968
ceavila3@miners.utep.edu

Amritam Sarcar
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
amritams@microsoft.com

Yoonsik Cheon and Cesar Yeep
Dept. of Computer Science
University of Texas at El Paso
El Paso, TX 79968
{ycheon@, ceyeep@miners}.utep.edu

***Abstract*—There are many benefits of checking design constraints at runtime—for example, automatic detection of design drift or corrosion. However, there is no comparative analysis of different approaches although such an analysis could provide a sound basis for determining the appropriateness of one approach over the others. In this paper we conduct a comparative analysis and evaluation of different constraint checking approaches possible for the Object Constraint Language (OCL). We compare several approaches including (1) direct translation to implementation languages, (2) use of executable assertion languages, and (3) use of aspect-oriented programming languages. Our comparison includes both quantitative metrics such as runtime performance and qualitative metrics such as maintainability of constraint checking code. We found that the implementation language-based approaches perform better in terms of memory footprint and runtime overheads but the other approaches are more appealing in terms of maintainability.**

***Keywords*-pre and postconditions; runtime constraint checking; AspectJ; JML; OCL**

I. Introduction

A recent trend in software development is a shift of focus from writing code to building models [1]. The idea is to systematically generate an implementation from a model through a series of transformations. A key requirement of this model-driven development is the availability of a precise model to generate working code from it. A formal notation such as the Object Constraint Language (OCL) [2] can play an important role to build such a precise model because the most popular modeling language, UML [3], lacks sufficient precision to enable complete code generation; OCL is a textual, declarative notation to specify constraints or rules that apply to UML models. Modeling and specifying design constraints explicitly is also said to improve reasoning of software architectures and thus their qualities [4].

Besides static reasoning, formally specified design constraints such as OCL constraints can be checked at runtime, and there are many benefits of checking design constraints at runtime. For example, it can detect when an implementation deviates from its design, often called *design corrosion* or *drift* [5] [6]. Design corrosion is said to be proportional to the development and maintenance time and occurs when the initial design of software gets modified to accommodate new or changed requirements or to correct defects. It also occurs as the result of code hacks and workarounds, a common practice of software maintenance. Runtime constraint checking can also facilitate automating program or conformance testing by allowing constraints to be used as test oracles [7].

Several different approaches are possible for checking design constraints such as OCL constraints against implementations at runtime. The most common approach is to translate constraints directly to an implementation language by coding a constraint checker in that language and making it part of the implementation [8]. Constraints can also be translated to executable assertions if the implementation language provides an assertion facility such as an `assert` statement or if it has a separate assertion languages [9] [10]. Yet another possibility is to apply aspect-oriented programming to modularize constraint checking code by implementing constraint checking as a crosscutting concern (see Section III-C for details) [11] [12].

We expect that each of the aforementioned approaches have its strengths and weaknesses. In this paper we conduct a comparative analysis of these approaches in order to determine appropriateness of one approach over the others. In our study we use OCL as the constraint specification language and Java as the implementation language. Our analysis will involve applying different approaches to a common set of OCL constraints and recording a set of metrics from each application. For the comparison we will use both quantitative metrics such as runtime speed and memory usage and qualitative metrics such as maintainability of constraint checking code. We will consider various types of OCL constraints such as class invariants, operation pre and postconditions.

The remainder of this paper is structured as follows. In Section II we briefly explain OCL by introducing example constraints to be used throughout this paper. In Section III we describe in detail four different constraint checking approaches by applying them to the example constraints. In Section IV we first describe the case study that we performed

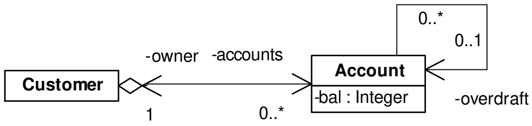

Figure 1. UML class diagram

to compare the approaches and then analyze the results from the case study. In Section V we conclude this paper with a summary of our findings.

II. Background on OCL

The Object Constraint Language (OCL) [2] is a textual, declarative notation to specify constraints or rules that apply to UML models. OCL can play an important role in model-driven software development because UML lacks sufficient precision to enable the transformation of a UML model to complete code. In fact, it is a key component of OMG's standard for model transformation for the model-driven architecture [13].

A UML diagram alone cannot express a rich semantics of and all relevant information about an application. The diagram shown in Figure 1, for example, is a UML class diagram for bank accounts. A customer can own several accounts, and an account can be linked to another account for overdraft protection. However, the class diagram doesn't express the fact that an account cannot be linked to itself for overdraft protection. It is very likely that a system built based only on diagrams alone will be incorrect. OCL allows to precisely describe this kind of additional constraints on the objects and entities present in a UML model. It is based on mathematical set theory and predicate logic and supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. The above-mentioned fact, for example, can be expressed in OCL as follows.

```
context Account
  inv: self <> overdraft
```

This constraint, called an *invariant*, states a fact that should be always true in the model. The keyword `self` denotes the object being constrained by an OCL expression, called a *contextual instance*; in this case it is an instance of the `Account` class. The invariant says that an account cannot be equal to its overdraft protection account. It is also possible to specify the behavior of an operation in OCL. For example, the following OCL constraints specifies the behavior of an operation `Customer::addAccount` by writing a pair of predicates called *pre* and *postconditions*.

```
context Customer::addAccount(acc: Account): void
  pre: not accounts->includes(acc)
  post: accounts = accounts@pre->including(acc)
```

The pre and postconditions states that, given an account not already owned by a customer, the operation should insert the account to the set of accounts owned by the customer. The postfix operator `@pre` denotes the value of a property (`accounts`) in the pre-state, i.e., just before an operation invocation. The constraints are written using OCL collection operations such as `includes` and `including`; the `includes` operation tests whether an object is contained in a collection, and the `including` operation adds an element to a collection.

OCL provides a few other constructs. The `init` construct specifies the initial value of an attribute or association end. The `def` construct introduces a new attribute or query operation to a UML model such as a class diagram. It also specifies the value of the new attribute or the return value of the new operation. The `derive` construct specifies the value of a derived attribute or association end, and the `body` construct defines the result of a query operation.

III. Runtime Constraint Checking

Several different approaches are possible for checking design constraints against implementations at runtime. For example, Froihofer et al. reviewed and evaluated different constraint validation approaches for Java [14]. They discussed handcrafted approaches, code instrumentation using OCL and JML [15], aspect-oriented programming, proxy implementations, CORBA, and EJBs. Each approach has its own advantages and disadvantages such as runtime overhead that ranges from a factor of two to more than one hundred. In this paper we focus on approaches available for OCL by considering OCL-specific features and consider only those approaches that do not require external or separate constraint checking monitors. We study the following three approaches classified by the target language to which OCL constraints are translated or in which the checking code is written.

- *Implementation languages.* This is the most widely-used approach and maps OCL constraints to an implementation language in that a constraint checker is written in that language and becomes part of the implementation (see for example [8]). If the implementation language supports an assertion facility such as the `assert` macro or statement, constraints can also be translated to executable assertions.
- *Assertion languages.* Some programming languages such as Eiffel support class invariants and operation pre and postconditions as built-in language constructs called *design-by-contract* [16]. Design-by-contract is not a formal part of Java, but there are several extensions or tools to support it for Java [15] [17]. OCL constraints can be translated to design-by-contract assertions [9] [10].
- *Aspect-oriented languages.* If an implementation language has an aspect-oriented extension, e.g, AspectJ for Java, it can be used to implement constraint checking code [11] [12] [18]. Constraint checking is viewed as a crosscutting concern and implemented as a so-called

aspect that resides in a separate module and *advises* the implementation code (see Section III-C below).

In the following subsections we explain each of the aforementioned approaches in detail using the OCL examples introduced in Section II.

A. Translating to Implementation Languages

1) Using Programming Language Statements: In this approach one injects hand-crafted checking code to an implementation. For each OCL constraint, one has to decide appropriate checking points and then translate the constraint to programming language statements. For example, a class invariant should be checked at the end of a constructor execution and before and after the execution of every method because a constructor has to establish the class invariant and every method has to preserve it. As an example, let us consider the `addAccount` operation of the `Customer` class introduced in Section II. Here is a possible implementation of the operation in Java.

```
public void addAccount(Account acc) {
  // check invariant at pre-state if any
  // check precondition
  if (accounts.contains(acc))
     throw new OclError("Precondition violation");
  // calculate accounts@pre
  Set<Account> accsPre = new HashSet<Account>(accounts);

  accounts.add(acc);
  acc.setOwner(this);

  // check postcondition
  accsPre.add(acc);
  if (!accounts.equals(accsPre))
     throw new OclError("Postcondition violation");
  // check invariant at post-state if any
}
```

As shown, the method body is wrapped with constraint checking code that checks the pre and postconditions and the class invariant as well in the pre and post-states.

In practice one would prefer to have a separate constraint checking method for each OCL constraint rather than embedding the checking code to the method body. This will modularize constraint checking code by eliminating duplicate code such as the invariant checking code. It will also facilitate reuse of constraint checking code and support for constraint inheritance; for example, to support the inheritance of an invariant, one only needs to make the invariant checking method of the subclass to call that of the superclass.

The main shortcoming of this approach is that there are a lot of manual work involved, such as translating OCL constraints to programming language statements and implementing the supporting infrastructure (e.g., one for constraint inheritance). Additionally, the resulting code may not be maintainable (refer to Section IV for an analysis).

2) Using Assertion Facilities: This approach is similar to the previous one except that OCL constraints are now translated to executable assertions of the implementation language. For example, the following is the `addAccount` method with OCL constraints translated to Java `assert` statements.

```
public void addAccount(Account acc) {
  assert !accounts.contains(acc) : "Precondition";
  Set<Account> accsPre = new HashSet<Account>(accounts);

  accounts.add(acc);
  acc.setOwner(this);

  accsPre.add(acc);
  assert accounts.equals(accsPre) : "Postcondition";
}
```

As in the previous approach, one has to determine appropriate constraint checking points and manually translate the constraints. However, one advantage of this approach is its ability to selectively enable or disable assertions; in Java, for example, one can control assertions at various granularities by using command-line switches.

B. Using Assertion Languages

In this approach, OCL constraints are translated to executable assertions such as design-by-contract. There are several extensions to Java that support design-by-contract [14]. For example, the Java Modeling Language (JML) [15] [17] is a formal interface specification language for Java to document the behavior of Java classes and interfaces, and a significant subset of JML is executable. The following code shows the `addAccount` method annotated with JML specifications.

```
/*@ public model JMLObjectSet accSet;
  @ private represents accSet
  @    = JMLObjectSet.convertFrom(accounts);
  @*/

/*@ requires !accSet.has(acc);
  @ ensures accSet.equals(\old(accSet.insert(acc)));
  @*/
public void addAccount(Account acc) { /* ... */ }
```

As shown, JML annotations are enclosed in special comments such as `/*@ ... @*/` and precede the Java declarations such as method declarations that are being annotated.[1] Method pre and postconditions follow the keywords **`requires`** and **`ensures`**, respectively. The JML-specific `\old` expression denotes the pre-state value of its argument. An interesting feature of JML is that it provides a built-in support for writing abstract specifications [19]. For example, the above pre and postconditions are written in terms of a specification-only variable `accSet` of which value is given as a mapping from a program variable `accounts`. This way of writing assertions have several advantages; for example, such assertions are less affected by implementation changes and do not expose implementation details such as a private field `accounts`. Another strength of using assertion languages is that OCL constraints are often directly mapped to assertions. This is particularly noticeable when translating OCL constraints consisting of iterator operations such as `forAll` and

[1] JML specifications can also reside in separate specification files.

`exists` because similar sorts of quantifiers are supported in JML.

C. Using Aspect-Oriented Programming Languages

Aspect-oriented programming is a new programming paradigm to address in a modular way so-called *crosscutting concerns* such as logging that have to be implemented in multiple program modules. The key idea is to denote a set of execution points, called *join points*, and introduce additional behavior, called an *advice*, at the join points. AspectJ [20] is an aspect-oriented extension for Java and provides built-in language constructs for join points and advices. OCL constraints can be systematically translated to AspectJ code to check at runtime the conformance of a Java implementation [11] [12] [18]. For example, the following AspectJ code checks the pre and postconditions of the `addAccount` operation.

```
public privileged aspect CustomerChecker {
  pointcut addAcountExe(Customer c, Account a):
    execution(void Customer.addAccount(Account))
      && this(c) && args(a);

  void around(Customer c, Account a): addAcountExe(c, a) {
    assert !c.accounts.contains(a) : "Precondition";
    Set<Account> accsPre = new HashSet<Account>(c.accounts);
    proceed(c, a);
    accsPre.add(a);
    assert c.accounts.equals(accsPre) : "Postcondition";
  }
}
```

The pointcut declaration designates a set of execution points and optionally exposes certain values at these execution points. For example, the pointcut `addAccountExe` denotes execution of the `addAccount` method and exposes the receiver (c) and the argument (a). The **`around`** keyword introduces an advice that wraps around a join point and can potentially replace it; there are also **`before`** and **`after`** advices. The above advice first checks the precondition by referring to the values exposed by the pointcut, proceeds to continue with the normal flow of execution at the join point (as indicated by the **`proceed`** keyword), and finally checks the postcondition. If the `Customer` class is compiled with the above aspect, every execution of the `addAccount` method will be checked against the pre and postconditions. The aspect-oriented approach has several advantages over the previous approaches. For example, the constraint checking logic is completely separated from the implementation, and the implementation modules are oblivious of the constraint checking code, even its existence. Thus, constraint checking code can be easily added or removed from the implementation. It will also enable runtime checks to be applied to different implementations of the same design and be selectively enabled or disabled, for example, for production code.

IV. Comparison

To find out the strengths and weaknesses of the approaches explained in the previous section, we compare them both quantitatively and qualitatively. For the comparison we use the following quantitative metrics.

- Source code size. We measure and compare this because it indicates the amount of work needed to implement constraint checking code. We also measure the number of source code lines needed to translate one line of a constraint.
- Bytecode size. This is one factor that determines the memory footprint of a program and thus may be important for certain systems like consumer electronics and embedded systems where low-memory-footprint programs are required.
- Dynamic memory usage. This is another factor that determines the memory footprint of a program.
- Execution time. This may be one of the most important criteria for selecting a constraint checking approach, especially for use in production code.

We also compare the approaches qualitatively using such criteria as easiness of translation, support for automation, and maintainability of checking code. In the following subsections we first describe the case study that we performed for the comparison and then analyze the comparison results.

A. Case Study

We performed a case study by using an open-source Java program that has a formal UML model including OCL constraints. The use of an open-source program eliminates subjectivenss during the experiment and makes the case study more realistic. The OCL standard specification defines several collection types such as Collection, Set, OrderedSet, Bag, and Sequence, and the behavior of each type is formally specified in OCL [21]. There are Java implementations of the OCL collection types [9] [22], and for our case study we used the one included in the Dresden OCL Toolkit [22]. This implementation supports all the collection operations specified in the standard except for iterator operations such as `forAll` that take OCL expressions as parameters and work on each element of a collection. The standard specifies 336 lines of OCL constraints, most of which are postconditions, and the implementation has 87 methods and 1781 lines of source code including comments.

We manually translated OCL constraints to runtime checks using each of the approaches. For the two Java-based approaches we directly modified the source code to insert assertion checking code. For the JML-based approach we also changed the source code to add JML annotations. For the AOP-based approach, however, instead of modifying the Java source code we introduced one AspectJ aspect for each Java class, responsible for checking all the constraints specified for that class. We next devised a suite of test data for each collection type and measured the runtime performance of each approach. The test suite also showed that all approaches are equally effective in detecting constraint

Table I
SOURCE CODE SIZE

	OCL	Src*	CC	CC/Src	CC/OCL
Stmt	336	1781	1025	0.58	3.05
Asrt	336	1781	401	0.22	1.19
JML	336	1781	330	0.19	0.98
AOP	336	1781	1257	0.71	3.74

*Src: Java code; CC: constraint checking code; size in LOC

Table II
BYTECODE SIZE

	All (kb)*	CC (kb)	CC/Src	CC/OCL
Stmt	42	20	0.91	0.06
Asrt	34	12	0.55	0.04
JML	307	285	12.96	0.85
AOP	71	49	2.22	0.15

*The columns show the size of base code plus checking code, the size of checking code, the ratio of checking code over base code (22kb), and the ratio of checking code over OCL lines (336 lines), respectively.

violations at runtime; it revealed several errors both in the implementation and in the constraints themselves [12].

B. Results

1) Source Code Size: The source code size is an important metric because it indicates the amount of work needed. Table I shows the measurement of source code size for each approach, given in the number of source code lines. It also shows the average number of source code lines needed to translate one line of OCL constraints (see the CC/OCL column). As expected, JML is superior in this metric because it supports similar language constructs as OCL including invariants, pre and postconditions, and quantifiers, and thus most OCL constraints can be directly translated to JML specifications. The AOP approach requires the most work because one has to not only translate OCL constraints to Java statements but also introduce AOP-specific declarations such as pointcuts, advices, and aspects.

2) Bytecode Size: Table II shows the bytecode size of constraint checking code.[2] The Java `assert` statement-based approach produces the most compact bytecode. It produces about 24 times more compact bytecode than the JML approach and requires on average about 21 times less bytecode per line of OCL constraints. It is interesting to learn that the bytecode size is not always proportional to the source code size. This is perhaps because both AspectJ and JML compilers have to produce code for a runtime support framework—for example, for dynamic pointcut resolution and for specification inheritance. Furthermore, the particular JML compiler (`jml4c`) that we used for this case translates quantifiers to inner classes, which brings an additional overhead on bytecode size [23]; more than half of the translated JML assertions were quantified expressions.

[2]The bytecode sizes for JML and AOP don't include those of runtime libraries such as `jml4rt.jar` and `aspectjrt.jar`.

Table IV
EXECUTION TIME

	No. of calls		CPU time	
	Number	Overhead	Sec	Overhead
Base	2101	0	0.04	0
Stmt	5842	1.78	0.11	1.98
Asrt	3124	0.49	0.09	1.09
JML	45646	20.73	2.99	79.13
AOP	14869	6.08	0.70	17.75

3) Dynamic Memory Usage: Table III shows the dynamic memory requirement for each approach, obtained using the Eclipse profiling tools. It shows the number of live instances, active size in bytes, the total number of instances, and the total size in bytes; a live instance is an instance that is alive, i.e., not garbage collected. The table also shows the memory overhead for each approach. The JML-based approach requires eight times more heap storages than the base code, and for other approaches the memory overhead is negligible. We suspect that this is because the JML compiler translates quantified assertions to inner classes.

4) Execution Time: Table IV shows the number of method calls and the CPU time required to run the test suite by each approach, along with the overhead due to constraint checking. As shown, the Java-based approaches outperform both the JML and the AOP-based approaches; for example, the JML-based approach is about 27 to 38 times slower than the Java-based approaches and requires 80 times more CPU time than the base code. This may be explained in part by the huge number of additional method calls introduced by the constraint checking code; JML translates each assertion to a separate assertion checking method and uses Java's reflection facility to support inheritance of specifications, e.g., to inherit specifications from the abstract superclass Collection. We also learned that application characteristics influence the execution times differently for different approaches; for example, both the AOP and the Java statement-based approaches require the longest execution time for the Collection class, the Java assert-based approach for the Set class, and the JML approach for the Sequence class.

5) Summary of Overheads: Table V and Figure 2 summarize the overheads of runtime constraint checking. The use of languages such as JML and AspectJ introduces significant runtime overheads on CPU time and dynamic memory storage. For example, programs with JML annotations require 84 times more CPU time and 13 times more heap storage than those without JML annotations. However, the increases of source and bytecode sizes are relatively negligible compared to runtime overheads. In summary, the Java-based approaches outperform the other approaches when considering only runtime overheads.

6) Qualitative Comparison: The case study also allowed us to compare the four approaches qualitatively using such criteria as translation easiness and maintainability of check-

Table III
DYNAMIC MEMORY USAGE

	Live instances		Active size		Total instance		Total size	
	Number	Overhead	Byte	Overhead	Number	Overhead	Byte	Overhead
Base	248	0	5552	0	581	0	10880	0
Stmt	337	0.26	6976	0.20	589	0.01	11040	0.01
Asrt	316	0.22	6640	0.16	590	0.02	11024	0.01
JML	447	0.45	47256	0.88	705	0.18	67576	0.84
AOP	461	0.46	8936	0.38	597	0.03	11144	0.02

Table V
SUMMARY OF CONSTRAINT CHECKING OVERHEADS

	Source	Bytecode	Memory	CPU Time
Stmt	0.58	0.91	0.01	1.98
Asrt	0.22	0.55	0.01	1.09
JML	0.19	12.96	0.84	79.13
AOP	0.71	2.22	0.02	17.75

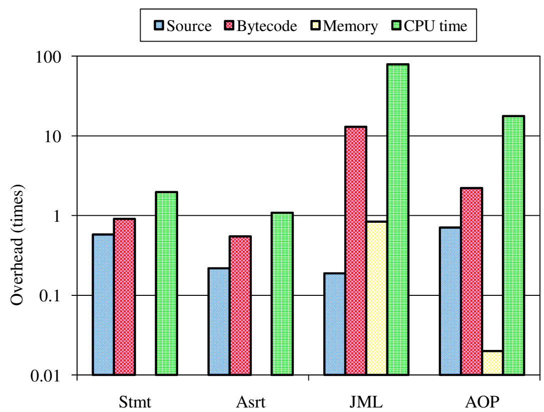

Figure 2. Summary of overheads

Table VI
QUALITATIVE COMPARISON

	Stmt	Asrt	JML	AOP
Translation	×	△	○	×
Automation	×	×	○	○
Readability	×	△	○	×
Reusability	×	×	○	○
Controllability	×	△	○	○
Maintainability	×	×	○	△
Maturity	○	○	×	△

○: good; △: fair; ×: bad

ing code, and Table VI summarizes the comparison result. Most OCL constructs and expressions were directly mapped and translated to JML specifications[3]. This was particularly noticeable for OCL expressions containing iterator operations such as `forAll` and `exists`, as most OCL iterator operations can be mapped to JML quantifiers. In other approaches, such expressions were translated to sequences of Java statements composed of loop statements. Regarding support for automation, there are OCL translation rules defined for JML and AspectJ along with a support tool for the AspectJ translation [9] [10] [22]. We found that it is a lot easier to read and understand constraint checking code when constraints are expressed in assertions such as Java `assert` statements and JML annotations rather than Java code statements. JML specifications and AspectJ source code are modular in that they can reside in separate specification or source code files and the base code—the module being annotated in JML or advised in AspectJ—doesn't depend on them. This has several benefits including reusability, controllability, and maintainability. The same JML specification or AspectJ code can be used for different implementations or versions of the same design; it is reusable and plug-and-playable. It is also easy to selectively turn on and off constraint checking by simply recompiling program modules; in JML, it is also possible to enable assertions based on their kinds such as invariants and method pre and postconditions. JML specifications and AspectJ code are more maintainable because they can better accommodate changes both in OCL constraints and checking code itself; they are not tangled with nor scattered over the base code. Regarding maturity of technology, JML is a research language still being developed and lacks stable support tools, and as a relatively new technology AOP is not widely accepted or used yet.

[3]However, there are a few rarely-used OCL operators such as message sending that are hard to express in JML because JML doesn't have built-in support for them.

V. CONCLUSION

We explained four different approaches for translating OCL constraints to runtime checks: (1) using implementation languages such as Java, (2) using built-in assertion facilities such as the `assert` statement, (3) using assertion or design-by-contract languages such as JML, (4) using aspect-oriented programming language such as AspectJ. We then compared these approaches critically through a case study. We learned that the first two approaches based on implementation languages are most efficient in terms of runtime performance such as CPU time and heap storage. However, our qualitative comparison favored the other two approaches. For example, OCL constraints, in most cases, can be directly translated to JML annotations, and the resulting JML specifications are easy to read and understand. There are translation rules from OCL to JML and AspectJ.

JML specifications and AspectJ code are better modularized and thus reusable, plug-and-playable, controllable, and maintainable. In summary, the first two approaches may be a better choice for the use of constraint checking in production code if memory footprint or runtime speed is an important concern. On the other hand, the other two approaches may be more appealing for the development use (e.g., testing and debugging) of constraint checking where concerns such as accommodation for changes are more important.

Acknowledgment

The work of the authors was supported in part by NSF grants CNS-0707874 and DUE-0837567.

References

[1] A. W. Brown, "Model driven architecture: Principles and practice," *Software and System Modeling*, vol. 3, no. 4, pp. 314–327, Dec. 2004.

[2] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison-Wesley, 2003.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Addison-Wesley, 2004.

[4] A. Tang and H. van Vliet, "Modeling constraints improves software architecture design reasoning," in *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 and European Conference on Software Architecture 2009 (WICSA/ECSA 2009*, 2009, pp. 253–256.

[5] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[6] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

[7] Y. Cheon and C. Avila, "Automating Java program testing using OCL and AspectJ," in *ITNG 2010: 7th International Conference on Information Technology: New Generations, April 12-14, 2010, Las Vegas, NV*. IEEE Computer Society, 2010, pp. 1020–1025.

[8] H. Hussmann, B. Demuth, and F. Finger, "Modulr architecture for a toolset supporting OCL," in *UML 2000 — The Unified Modeling Language, Advancing the Standard, York, UK, October 2000*, ser. LNCS, A. Evans, S. Kent, and B. Selic, Eds. Springer-Verlag, 2000, vol. 1939, pp. 278–293.

[9] C. Avila, G. Flores, and Y. Cheon, "A library-based approach to translating OCL constraints to JML assertions for runtime checking," in *International Conference on Software Engineering Research and Practice, July 14-17, 2008, Las Vegas, Nevada*, 2008, pp. 403–408.

[10] A. Hamie, "Translating the Object Constraint Language into the Java Modeling Language," in *Proceedings of the ACM Symposium on Applied Computing, Nicosia, Cyprus, March 14 -17, 2004*, 2004, pp. 1531–1535.

[11] L. C. Briand, W. J. Dzidek, and Y. Labiche, "Instrumenting contracts with aspect-oriented programming to increase observability and support debugging," in *Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, September 25-30, 2005*, Sep. 2005, pp. 687–690.

[12] Y. Cheon, C. Avila, S. Roach, and C. Munoz, "Checking design constraints at run-time using OCL and AspectJ," *International Journal of Software Engineering*, vol. 2, no. 3, pp. 5–28, Dec. 2009.

[13] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, Jan. 2003.

[14] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka, "Overview and evaluation of constraint validation approaches in Java," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 313–322.

[15] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Soft. Eng. Notes*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[16] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[17] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, Jun. 2005.

[18] W. J. Dzidek, L. C. Briand, and Y. Labiche, "Lessons learned from developing a dynamic OCL constraint enforcement tool for Java," in *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, October 2-7, 2005*, ser. LNCS. Springer-Verlag, 2006, vol. 3844, pp. 10–19.

[19] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, "Model variables: Cleanly supporting abstraction in design by contract," *Software—Practice & Experience*, vol. 35, no. 6, pp. 583–599, May 2005.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, ser. LNCS, J. L. Knudsen, Ed. Berlin: Springer-Verlag, Jun. 2001, vol. 2072, pp. 327–353.

[21] OMG, *UML 2.0 OCL Specification*. Object Management Group, Oct. 2003, available from http://www.omg.org/docs/ptc/03-10-14.pdf (retrieved on Feb. 23, 2010).

[22] B. Demuth and C. Wilke, "Model and object verification by using Dresden OCL," in *Proceedings of the Russian-German Workshop Innovation Informa-tion Technologies: Theory and Practice, July 25-31, Ufa, Russia, 2009*, 2009, pp. 687–690.

[23] A. Sarcar and Y. Cheon, "A new Eclipse-based JML compiler built using AST merging," Department of Computer Science, The University of Texas at El Paso, Tech. Rep. 10-08, Mar. 2010.