Enforcing Information Hiding in Interface Specifications: A Client-Aware Checking Approach

Henrique Rebêlo and Gary T. Leavens

Universidade Federal de Pernambuco, PE, Brazil
hemr@cin.ufpe.br

University of Central Florida, Orlando, FL, USA
leavens@eecs.ucf.edu

Abstract

Information hiding is an established principle that controls which parts of a module are visible to non-privileged and privileged clients (e.g., subclasses). This aids maintenance because hidden implementation details can be changed without affecting clients. The benefits of information hiding apply not only to code but also to other artifacts, such as specifications. Unfortunately, contemporary formal interface specification languages and their respective runtime assertion checkers (RACs) are inconsistent with information hiding rules because they check assertions in an overly-dynamic manner on the supplier side. We explain how overly-dynamic RACs compromise information hiding and how our client-aware checking technique allows these RACs to use the privacy information in specifications, which promotes information hiding.

Categories and Subject Descriptors D.2.4 [*Software*]: Program Verification—Programming by contract; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Assertions, Invariant, Pre- and postconditions, Specification techniques

General Terms Languages, Verification

Keywords Information hiding, interface specification languages, runtime assertion checking, JML language, AspectJML language

1. Introduction

Information hiding [26] (also known as black-box abstraction) is a widely accepted principle that aids software development. It advocates that a module should expose its functionality but hide its implementation behind an interface. This supports modular reasoning and independent evolution/maintenance of the hidden parts of a module. If programmers have carefully chosen to hide those parts "most likely" to change [26], most changes, in the hidden implementation details, do not affect module clients.

Information hiding and its benefits apply not only to code but also to other artifacts, such as documentation and specifications [22]. In this paper, we focus on formal interface specifications and their runtime assertion checking (RAC) tools. Formal interface specifications include contracts written in Eiffel [24], the Java Modeling Language (JML) [21], Spec# [3], and Code Contracts [9].

However, all the mentioned interface specification languages and their RAC tools violate the information hiding principle in some sense; in the following we discuss these issues in detail.

1.1 A Running Example

Figure 1 illustrates our running example. It is a simple delivery service system, written in Java-like style with pre- and postconditions, which manages package delivery [25, pp.100-107][1]. As illustrated, the classes `Package`, `Destination`, and `Courier` are implemented by Alice. The `Courier` is a client of both `Package` and `Destination` classes. In the figure we use the notations **`@pre`** and **`@post`** to introduce pre- and postcondition specifications for each method. An omitted precondition is shorthand for **`@pre true`**. We use **`@ret`** to stand for the result of a method in its postcondition. These notations have their counterparts in contemporary interface specification languages such as Eiffel [24], JML [21], Spec# [3], and Code Contracts [9].

1.2 Information Hiding in Specifications

Leavens and Müller [22] present rules for information hiding in specifications for Java-like languages. Normally, information hiding [26] controls which parts of a class are visible to clients. This aids maintenance because hidden implementation details can be changed without affecting clients. For example, one should keep pre- and postconditions that describe implementation details hidden, so that when such implementation details are changed, clients are not affected.

In terms of information hiding, the example presented in Figure 1 has a weakness. The field `weight` declared in `Package` is public,[2] not private. Suppose that Alice now decides to change this field to be private. When doing this Alice must ensure that the clients continues to work as before.

However, if we have a public client, such as the one written by Cathy in Figure 2, of the method `setWeight` of `Package`, and

MODULARITY Companion'15, March 16–19, 2015, Fort Collins, CO, USA
Copyright 2015 ACM 978-1-4503-3283-5/15/03...$15.00
http://dx.doi.org/10.1145/2735386.2736750

[1] This code would be improved if `weight` and `address` fields were declared with protected or private visibility. But this would require using additional features related to information hiding that are discussed in the next subsection. So for simplicity we use a less elegant and less secure approach here.

[2] For simplicity, we consider only the class Package, although similar considerations apply to the class `Destination`, which also has a public field.

```
                                        // written by Alice

public class Package {                  public class Destination {              public class Courier {
  /* intensionally public */              /* intensionally public */           public int deliver(Package p,
  public double weight;                   public String address;                                 Destination d)
                                                                                    @post @ret <= 3; // hours
  public void setWeight(double weight)    public void setAddress(String address)  {
    @pre weight <= 5;                       @pre address != null;                   /* delivery service */
    @post this.weight == weight;            @post this.address.equals(address);   }
  {                                       {                                     }
    this.weight = weight;                 }
  }                                       public String getAddress()
  public double getWeight()                 @pre this.address != null;
    @pre this.weight <= 5;                  @post @ret.equals(this.address);
    @post @ret == this.weight;            {
  {                                         return this.address;
    return this.weight;                   }
  }                                     }
}
```

Figure 1. The pre- and postcondition specifications of the delivery service classes [25, pp.100-107].

```
// written by Cathy
public class ClientClass {
  public void clientMeth(Package p)
    { p.setWeight(5); }
}
```

Figure 2. Client code written by Cathy.

the method's implementation violates the postcondition defined in Figure 1, then Cathy would see a postcondition violation, referring to the private field `weight`. The problem is that such a field is hidden from public clients. Thus such contract violations, involving hidden fields, are not meaningful to all clients. As a consequence, clients confront an issue that the interface claimed to hide [15, 17].

In this context, Leavens and Müller's Rule 1 [22] says that specifications should not expose hidden members. This implies that the public postcondition of class `Package` must not mention the private field `weight`. The specification language JML [21] overcomes this problem by providing syntax for visibility modifiers [22] that is applied to various specification constructs. These modifiers allow one to specify a class's public (non-privileged client), protected (subclass), package (friend), and private (implementation) interfaces.

```
public class Package {
 //@ public model double weight;
 private double _weight;
 //@ private represents weight = _weight;

  /*@ public normal_behavior
    @    requires weight <= 5;
    @    ensures this.weight == weight;
    @ also
    @  private normal_behavior
    @    requires weight <= 5;
    @    ensures this._weight == weight;
    @*/
 public void setWeight(double weight) {
  this._weight = weight;
 }
 ...
}
```

Figure 3. The public and private specifications of the class `Package` with JML.

As an example, Figure 3 shows two JML specifications for the method `setWeight` in `Package`: a public specification case (useful for Cathy) and a private specification case for privileged clients (Alice in this case). The visibility issues are resolved using model fields [7, 20], which are specification-only fields that give an abstraction of some concrete state. In this example, the value of the public model field `weight` is determined directly by the value of the corresponding private field `_weight`. This relationship is specified by JML **`represents`** clauses. This **`represents`** clauses is private, since it mentions a private field.

The specification for each visibility level is written as a separate specification case, each of which contains requires and ensures clauses (for pre- and postconditions). The specification cases shown each start with a visibility modifier and the keyword **`normal_behavior`**. (The **`normal_behavior`** keyword indicates that the method must not throw exceptions when that specification case's precondition is satisfied.) The interpretation of such a specification case is that when its precondition is satisfied, the method's execution must finish normally in a post-state that satisfies that case's postcondition. These specification cases for a method are separated by the keyword **`also`** (which means that each specification case must be obeyed when its precondition holds [20]).

1.3 Problem: Information Hiding and RAC

Although there is a set of rules for information hiding in specifications [22] and although JML [21] supports visibility modifiers, existing runtime assertion checker (RAC) tools violate those information hiding rules during runtime assertion checking and error reporting.

The problem is the way contemporary RACs implement runtime assertion checks for method calls. Such RAC compilers operate by injecting code at the supplier side (see Figure 4), thus checking each method's precondition at the beginning of its code, and injecting code to check the method's postcondition at the end of its code. Since supplier-side checks do not take into account what kind of client (i.e., privileged or non-privileged) is calling a method, a RAC checks all the specifications regardless of visibility. Therefore, we say that a RAC that checks all the specifications at supplier side as *overly-dynamic*.

For example, consider again the client code (written by Cathy) in Figure 2. Consider further that the implementation of `setWeight` mistakenly increments the weight by 1. In this scenario, we got the following postcondition error in the classic JML RAC:

```
Exception in thread "main" org.jmlspecs.ajmlrac.runtime.
     JMLExitNormalPostconditionError:
by method Package.setWeight regarding code at
File "Package.java", line 15 (Package.java:15), when
  'this._weight' is 6.0
  'weight' is 5.0
  ...
```

```
public class Package {
 private double _weight;

  public void setWeight(double weight) {
   this.weight = weight;
    if (!checkNPost$setWeight$Package(weight))
     throw new NormalPostCondition();
  }
   public boolean checkNPost$setWeight$Package(double w)
   { return (this.weight == w) && (this._weight == w); }
}
```

Figure 4. `Package`'s postcondition instrumentation used by JML's RAC. This is similar to what is done in other RACs (excepting the use of specification inheritance).

As can be seen, in this error output, the private field `_weight` is mentioned (see the shadowed part). However, this field is not meaningful to clients, as it is not visible. So, supplier side checking in JML makes it difficult to implement client-oriented error reporting that only shows clients error reports that make sense at the call site. This problem also occurs in other specification languages like Eiffel [24], Spec# [3], and Code Contracts [9].

Indeed, some violations can also be unexpected due to the visibility rules, as according to the Leavens and Müller's rules [22], only the public specification cases (Figure 3) should be checked against the method call `p.setWeight(5)` in a public client.

1.4 Contributions to the State-of-the-art

This paper has two main contributions. The first is the explicit recognition of the visibility problems that plague contemporary RACs due to their overly-dynamic checking in compromising information hiding rules. Second, it contributes an improved form of runtime assertion checking that is consistent with information hiding. We introduce (in Section 2) the concept of *client-aware checking* (CAC), and show how it checks pre- and postconditions for OO programs in a way that it respects information hiding.

To the best of our knowledge, our approach is the first to explicitly identify and fix this problem related to overly-dynamic checking of contemporary RAC tools. Our CAC technique works by injecting code to check pre- and postcondition into clients at method call sites. Because code is injected at the site of each method call, it properly uses the visible specifications associated with the clients.

2. Client-Aware Checking (CAC)

Our goal is to ease proper checking of visibility rules in a RAC. We call our technique for doing this client-aware checking (CAC). CAC aims to runtime check contracts from the client's point of view. We describe CAC in more detail below.

The most important feature of CAC is that it checks method specifications on the client side of each call (i.e., at the call site). Doing this allows CAC to be consistent with information hiding rules, by checking only the visible pre- and postconditions for each call. This also avoids the not meaningfull error reporting that may arise from overly-dynamic RACs.

To see how these checks are made, consider Figure 5, which contains the public client written by Cathy (Figure 2) based on the `Package`'s JML specifications shown in Figure 3. We use the proof rule for method calls that allows one to derive

$$\{pre_m^T[\vec{a}/\vec{f}]\}\ p.m(\vec{a})\ \{post_m^T[\vec{a}/\vec{f}]\}$$

from a specification $(T \triangleright pre_m^T, post_m^T)$ associated with the receiver p's the static type T. (The notation $[\vec{a}/\vec{f}]$ means the substitution of the actual parameters $\vec{a}$ for m's formals $\vec{f}$.) An automated static verifier that uses weakest precondition semantics can modularly replace a call $p.m(\vec{a})$ by the sequence of "**assert** $pre_m^T[\vec{a}/\vec{f}]$; **assume** $post_m^T[\vec{a}/\vec{f}]$" [2].

A RAC will do runtime checks at each **`assert`** and **`assume`** statement. That is, in the CAC technique, a RAC injects runtime checks around each method call to check the pre- and postconditions of the statically-visible specifications for the call.

```
/* Client-side reasoning
   and instrumentation with CAC */
public class ClientClass {
  public void clientMeth(Package p) {
    //@ assert 5 <= 5;
    p.setWeight(5);
    //@ assume p.weight == 5;
  }
}
```

Figure 5. Example of Client-Aware Checking instrumentation for the client code written by Cathy (Figure 2).

Now if we consider again that the implementation of `setWeight` (Figure 1) mistakenly increments the weight by 1, we got the following postcondition error in our CAC implementation of JML:

```
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.
     JMLExitPublicNormalPostconditionError:
by method Package.setWeight regarding code at
File "Package.java", line 15 (Package.java:15), when
  'this.weight' is 6.0
  'weight' is 5.0
  ...
```

As observed, this time the error output does not mention the private (hidden) field `_weight` anymore. The public client now is aware of the specifications that are being checked during runtime. In other words, information hiding is working as expected and the client has all the benefits expected from runtime assertion checking. Also, CAC provides a specific postcondition violation for public postconditions (as shadowed above).

The private postcondition should be omitted from the error reporting since specifications of wider visibility should be refined by their counterparts of narrower visibility (hence public method preconditions should imply private preconditions, and private postconditions should be implied by public postconditions, when the public preconditions hold [8]). The reason for this is that a client would be surprised if they encountered assertion violation errors due to invisible assertions.

The benefits of information hiding enforcement by CAC is evident when, for example, Alice decides to change the name of the private field `_weight` (Figure 1) to a different one, and the public client Cathy (Figure 2) is not affected by such a change.

2.1 Implementation of CAC

We implemented the CAC technique in the AspectJML RAC compiler (ajmlc) [27, 32] which is available online at `http://www.cin.ufpe.br/~hemr/aspectjml/`. This is the first RAC to support CAC. The ajmlc compiler itself was both described in previous works [27, 32] and demonstrated [28, 29]. It takes input written in JML [21] and generates aspects [6, 10, 16, 18, 30] to check those specifications at runtime.

Figure 6 shows the **`after returning`** advice generated by the ajmlc compiler to check the private postconditions of method

```
/** Generated by AspectJML to check the private
  * normal postcondition of method setWeight. */
after (Package object$rac, double weight) returning():
 call(void Package.setWeight(double))
 && within(Package)
 && target(object$rac) && args(weight) {
  boolean rac$b = (!(weight <= 5.0D) ||
        (object$rac._weight == weight));
  JMLChecker.checkNormalPostcondition(rac$b, "...");
}
```

Figure 6. Generated after returning advice to check the private specifications of `Package.setWeight` in Figure 3.

`setWeight` defined in Figure 3.[3] The shaded part is an optimized version of the privacy checking code. The variable `rac$b` denotes the postcondition to be checked. This variable is passed as an argument to `JMLChecker.checkNormalPostcondition`, which checks such postconditions; if it is not true, then a postcondition error is thrown.

We rely on the AspectJ **`within`** pointcut in the generated advice (see the shadowed part in Figure 6). It guarantees the effective private postcondition is checked to all calls to method `setWeight` that lexically occurs from within `Package`.

For public postconditions, ajmlc uses the **`within`**`(*)` pointcut with a wildcard. This ensures that any call to the method `setWeight`, regardless its client visibility, must respect the specified public postcondition. Note that ajmlc generates a dedicated **`after returning`** advice for each JML specification case with different privacy modifiers.

3. Related Work

A crucial feature of modules is that they support *information hiding*. Essentially, this means that the module's declarations are sliced into a public module interface, which is visible to all clients of the module, and a private module body, which contains the module implementation and is hidden from clients. The public module interface's behavior can be specified using a formal interface specification language. In this context, the most related work is by Leavens and Müller [22]. They present rules for information hiding in specifications for Java-like languages. Although some of these rules are enforced during compile time in JML, during the runtime assertion checking they are not taken into account as described in Section 1.3. We complement their work with a precise runtime checking for JML information hiding features.

Aside from JML, as mentioned in the introduction section, there are several other interface specification languages, such as Eiffel [24], Spec# [3], and Code Contracts [9]. (The latter two are both interface specification languages for C#.)

Eiffel and Spec# have the same runtime checking problems that we discussed above for JML's Rac tool, since the implementation of JML and Spec# drew heavily on techniques pioneered by Eiffel.

Code Contracts [9] has an interesting relationship to our work: it also supports runtime checking at call sites. However, in Code Contracts only preconditions can be checked at call sites. With CAC, we can also check postconditions (and invariants) at call sites. In addition, Code Contracts do not support information hiding features. As a consequence, Code Contracts would not be able to properly check postconditions with different privacy modifiers.

The work by Findler and Felleisen [11–13] is closest in spirit to our CAC technique. Their work describes contract soundness for a language called Contract Java. In Contract Java a programmer can specify pre- and postconditions of methods defined in classes and interfaces. Their work is closely related to ours in the sense that the translation rules for Contract Java also inject runtime checking code at call sites; that is, they perform client-side checking. Hence their work is a precedent for our work on CAC.

However, Findler and Felleisen's work is primarily concerned with enforcing behavioral subtyping and presenting the novel idea of soundness of contract checking. Thus, unlike our work, their work does not consider privacy levels in specifications nor enforces information hiding as we do with CAC.

Similar to our CAC/AspectJML language, there are several works in the literature that instrument DbC with AOP [10, 16, 23, 30]. Kiczales opened this research avenue by showing a simple precondition constraint implementation in one of his first papers on AOP [16]. After that, other authors explored how to implement and separate the DbC concern with AOP [10, 16, 23, 30, 31]. All these works offer common templates and guidelines for DbC aspectization.

We go beyond these works by showing how to use AOP/AspectJ to enforce information hiding in JML contracts at source code. We believe that this can be exploited by these works specially if they consider to use client side instrumentation as we do here.

Pipa [34] is a design by contract language tailored for AspectJ. As with CAC/AspectJML, Pipa is an extension to JML. However, unlike our work, Pipa does not enforce information hiding in the AspectJ programs specified with JML. Hence, our client-aware checking approach is certainly an open issue that could be added to enhance the expressiveness of the Pipa language.

There are several other advanced runtime assertion checking approaches, such as trace-based [1, 4, 5] and history-based assertions [14, 19, 33], that we believe they can be benefited from our client-aware checking approach. For example, temporaljmlc [14] is a JML extension that performs runtime assertion checking of temporal properties. Since this tool is JML based, our client-aware checking approach is a natural candidate to be included in temporaljmlc. Another possibility is to extend our CAC technique with temporal properties implemented directly in AspectJ, as done by Stolz and Bodden [33].

4. Expected Feedback

At the most general level, we are interested in getting feedback on the importance of information hiding at source code, with explicitly syntactic constructs like those of JML, and runtime (by using runtime assertion checking tools). We are really interested in discussing this with the programming languages, software engineering, and modularity communities at large to see if the initial findings and impressions of this work is shared.

At a more specific level, we are looking forward to feedback on the strategy and implementation of client-aware checking (CAC).

Therefore, discussion on these topics would be of particular interest for the further development and improvement of client-aware checking.

5. Summary

Client-aware checking, CAC, enables consistent runtime assertion checking and modular reasoning in the presence of information hiding in specifications. CAC represents a change in the way the runtime checks are injected into code, since checks are placed in client code as opposed to being only done in supplier classes. Since runtime checks are injected at the point of method calls, the client's perspective on the called method can be taken into account, which allows runtime checking to be consistent with privacy modifiers used for information hiding.

[3] The ajmlc compiler provides a compilation option that prints all the checking code as aspects instead of weaving them.

6. Acknowledgments

We thank Rustan Leino, Mike Barnett, Peter Müller, Shuvendu Lahiri, and Tom Ball for discussions about these topics. Mike Barnett also gave us a demo of Code Contracts at MSR.

Special thanks to Mira Mezini and Ralf Lämmel for detailed discussions and for comments on an earlier version of these ideas.

We would like to thank David Naumann and Bertrand Meyer for interesting discussions during CBSoft 2012 (Natal/Brazil).

References

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th Annual ACM SIGPLAN OOPSLA'05*, pages 345–364, New York, NY, USA, 2005. ACM.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31:82–87, September 2005.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*. Springer-Verlag, 2005.

[4] E. Bodden, F. Chen, and G. Rosu. Dependent advice: A general approach to optimizing history-based aspects. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 3–14, New York, NY, USA, 2009.

[5] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative runtime verification with tracematches. *J. Log. and Comput.*, 20(3):707–723, June 2010.

[6] L. C. Briand, W. J. Dzidek, and Y. Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 687–690, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599, May 2005.

[8] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is ISU CS TR #95-20c, `http://tinyurl.com/s2krg`.

[9] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10. ACM, 2010.

[10] Y. A. Feldman et al. Jose: Aspects for design by contract80-89. *IEEE SEFM*, 0:80–89, 2006.

[11] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 1–15, New York, NY, USA, 2001. ACM.

[12] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 229–236, New York, NY, USA, 2001. ACM.

[13] R. B. Findler, M. Latendresse, and M. Felleisen. Object-oriented programming languages need well-founded contracts. Technical report, Department of Computer Science, Rice University, 2001.

[14] F. Hussain and G. T. Leavens. Temporaljmlc: A jml runtime assertion checker extension for specification and checking of temporal properties. In *Proceedings of the 8th IEEE SEFM'10*, pages 63–72, Washington, DC, USA, 2010. IEEE Computer Society.

[15] G. Kiczales. Beyond the black box: Open implementation. *IEEE Softw.*, 13(1):8–11, Jan. 1996.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with aspectj. *Commun. ACM*, 44:59–65, October 2001.

[17] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 481–490, New York, NY, USA, 1997. ACM.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[19] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In G. T. Leavens, C. Clifton, and R. Lämmel, editors, *Foundations of Aspect-Oriented Languages*, Mar. 2005.

[20] G. T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Z. Liu and H. Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, Nov. 2006. Springer-Verlag.

[21] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 2006.

[22] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, May 2007.

[23] C. V. Lopes, M. Lippert, and E. A. Hilsdale. Design By Contract with Aspect-Oriented Programming. In *U.S. Patent No. 06,442,750*, issued August 27, 2002.

[24] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[25] R. Mitchell, J. McKim, and B. Meyer. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[26] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[27] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm. Aspectjml: Modular specification and runtime checking for crosscutting contracts. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 157–168, New York, NY, USA, 2014. ACM.

[28] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm. Modularizing crosscutting contracts with aspectjml. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 21–24, New York, NY, USA, 2014. ACM.

[29] H. Rebêlo, G. T. Leavens, and R. M. Lima. Client-aware checking and information hiding in interface specifications with jml/ajmlc. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pages 11–12, New York, NY, USA, 2013. ACM.

[30] H. Rebêlo, R. Lima, and G. T. Leavens. Modular contracts with procedures, annotations, pointcuts and advice. In *SBLP '11: Proceedings of the 2011 Brazilian Symposium on Programming Languages*, 2011.

[31] H. Rebêlo, R. Lima, G. T. Leavens, M. Cornélio, A. Mota, and C. Oliveira. Optimizing generated aspect-oriented assertion checking code for jml using program transformations: An empirical study. *Sci. Comput. Program.*, 78(8):1137–1156, Aug. 2013.

[32] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, and M. Cornélio. Implementing Java modeling language contracts with AspectJ. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 228–233, New York, NY, USA, 2008. ACM.

[33] V. Stolz and E. Bodden. Temporal assertions using aspectj. *Electron. Notes Theor. Comput. Sci.*, 144(4):109–124, May 2006.

[34] J. Zhao and M. Rinard. Pipa: a behavioral interface specification language for AspectJ. In *Proceedings of the 6th FASE'03*, pages 150–165, Berlin, Heidelberg, 2003. Springer-Verlag.