Towards Client-Aware Interface Specifications

Henrique Rebêlo
Federal University of Pernambuco, Recife, PE, Brazil
hemr@cin.ufpe.br

Abstract

Runtime assertion checking (RAC) is a well-established technique for runtime verification of object-oriented (OO) programs. Contemporary RACs use specifications from the receiver's dynamic type when checking method calls. This implies that in presence of subtyping and dynamic dispatch features of object-oriented programming, these specifications differ from the ones used by static verification tools, which rely on the specifications associated with the static type of the receiver. Besides the heterogeneity problem, this also hinders the benefits of modular reasoning achieved by the notion of supertype abstraction. In this context, we propose a more precise runtime assertion checking for OO programs that better matches the semantics used in static verification tools. While we describe our approach, we discuss how it can be used to avoid the heterogenous semantics problem and among others.

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

General Terms Design, Languages, Verification

Keywords Modular Reasoning, Runtime Verification, Client-Aware Interface Specifications

1. Introduction

Object-oriented programming (OOP) has been presented for many virtues, of which we can emphasize subtyping and dynamic dispatch. Both are useful and problematic in relation to the procedural approach that OOP replaces. They are useful because one can abstract away details in the specifications of subtypes using the supertype ones. This allows variations in data structures and algorithms to be handled uniformly with subtype polymorphism. They are problematic for reasoning about object-oriented (OO) programs, because dynamic dispatch selects different methods depending on the exact runtime type of an object. For example, a dynamically dispatched method call such as `o.m()` requires a case analysis to deal with all possible dynamic types of `o`'s value. Hence, we need to re-specify or re-verify the method `m` whenever new subtypes are added to the program. However, such an approach is not modular, because it requires re-specifying or re-verifying existing code when the program is extended.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

In this context, Leavens and Weihl [7] proposed a strategy for modular reasoning, which they call "*supertype abstraction*". Such a strategy is modular in that it does not depend on receiver's dynamic type. For instance, What specification should one use to reason about a call, such as `o.m()`, given that the static type of `o` is `T`?. Based on the supertype abstraction technique, one should use the specification associated with the static type of `o` (`T` in this case) to reason about the correctness of a method call. As supertype abstraction does no depend on the `o`'s dynamic type, the method `m` does not need to be re-specified or re-verified when the existing subtypes of `T` are changed or when new subtypes are added to a program.

The benefits of the supertype abstraction idea are related to the Liskov's invited talk at OOPSLA 1987 [9]. Liskov stated an easily-remembered test for subtyping, also called Liskov Substitutability Principle (LSP), (p. 25): "if for each object `o1` of type `S` there is an object `o2` of type `T` such that for all programs `P` defined in terms of `T`, the behavior of `P` is unchanged when `o1` is substituted for `o2` then `S` is a subtype of `T`".

Problems. Although supertype abstraction is a helpful technique to reason about object-oriented programs, current runtime assertion checkers use specifications from the receiver's dynamic type when checking client method calls. As a consequence, to reason about the method call `o.m()`, one need to perform a case analysis with all possible dynamic types of receiver `o`. This approach hinders modular reasoning and raises other problems. To illustrate, consider the code in Figure 1 from the canonical figure editor example [6, 10]. We use JML [8] as our formal interface specification language for concreteness, but the problems and solution we present can also be exploited to other interface specification languages (e.g. Spec# [2]). In JML, annotation comments start with an at-sign (@) and specification cases for methods start with a visibility modifier and **normal_behavior**, both appear before the method's header. Preconditions are introduced by keyword **requires** and postconditions by **ensures**.

Figure 1 gives protected specifications for classes `Point` and `ScreenPoint`. In the class `Point`, the method `setX`'s precondition states that the argument `x` must be greater than or equal to zero. The postcondition ensures that the coordinate `Point.x` (the field `x` of `Point` class) is equal to the value of the argument `x`. The `Point`'s subclass `ScreenPoint` overrides the inherited `setX` method and provides an additional JML specification case that describes how the method behaves for arguments that do not satisfy the precondition of the inherited protected specification case (in JML the keyword **also** means that the specification of `Point.setX` is inherited to `ScreenPoint.setX`). Hence, when the argument `x` is less than zero (precondition) the inherited coordinate `Point.x` must be zero (postcondition).

Heterogeneous Semantics Problem. The first problem with runtime verification of specified OO programs is that the specifications used to check the correctness of a method call is based on its dynamic type, thus hindering modular reasoning (supertype ab-

```
1  package p;
2  class Fig {}
3  class Point extends Fig {
4   protected int x, y;
5   /*@ protected normal_behavior
6     @ requires x >= 0;
7     @ ensures this.x == x; @*/
8   public void setX (int x) {
9    this.x = x;
10  }
11 }
```

```
12 package p;
13 class ScreenPoint extends Point{
14  /*@ also
15    @ protected normal_behavior
16    @ requires x < 0;
17    @ ensures this.x == 0; @*/
18  public void setX (int x) {
19   if (x >=0) this.x = x;
20   else x = 0;
21  }
22 }
```

```
23 package p; // protected client
24 class clientClass1 {
25  void clientMeth1 (Point p) {
26   p.setX(-1);
27  }
28 }
29 package q; // public client
30 class clientClass2 {
31  void clientMeth2 (Point p) {
32   p.setX(-1);
33  }
34 }
```

Figure 1. Behavioral contracts for the figure editor [6, 10] using JML [8].

```
1  package p;
2  class Point extends Fig {
3   protected int x, y;
4   public void setX (int x) {
5    this.x = x;
6   }
7  }
```

```
8  package p;
9  // Behavioral Interface Specification
10 class Point extends Fig {
11  /*@ protected normal_behavior
12    @ requires x >= 0;
13    @ ensures this.x == x; @*/
14  public void setX (int x);
15 }
```

```
16 package p; // protected client
17 class clientClass1 {
18  void clientMeth1 (Point p) {
19   //@ assert -1 >= 0;
20   p.setX(-1);
21   //@ assume this.x == -1;
22  }
23 }
```

Figure 2. Formulation of Client-Aware Interface Specifications.

straction) and resulting in a heterogenous semantics in contrast to some verification tools, which are static type reasoning-based [4]. For example, consider the call to method setX on line 26 (Figure 1). The technique of supertype abstraction [7] uses the specification of the static type of the receiver to reason about such a call. Hence, since p's static type is Point, supertype abstraction tell us to reason about the call p.setX(−1) using the specification given on lines 5−7. As a result such a call violates the precondition (line 6) when passing −1 as argument to method setX.

However, by using the classical JML runtime assertion checker (RAC) [4], we got no precondition violation when the receiver p represents the dynamic type ScreePoint. This happens because the effective precondition used is the specifications given on lines 5−7 is joined with the specification on lines 14−17 (this give us the effective precondition (x >= 0) || (x < 0)). In other words, this problem happens because the instrumentation technique is done locally at the method declaration site. For instance, the JML RAC compiler (jmlc) uses an approach called *wrapper approach* [4]. This approach translates pre- and postcondition specifications into separate *assertion checking methods* which wraps the original method implementation with such assertion checking methods. Thus, all client calls now go to the wrapper method. In addition, the wrapper approach is responsible for calling corresponding assertion checking methods of supertypes if any. Because of that, the method call p.setX(−1) includes the specifications of type ScreePoint when the receiver p matches it (thus, going against supertype abstraction).

On the other hand, if we use the static checker ESC/Java2 [4] on the same method (call on line 26 in Figure 1), we can now detect the expected precondition violation based on the specifications of class Point (lines 5−7). Therefore, this causes another fundamental problem for program verification; the existing tools [4] use a heterogeneous semantics for program verification. For instance, the static checker is based on static type reasoning, whereas the runtime assertion checker is based on dynamic type reasoning.

Visibility Rules Checking Problem. Leavens and Müller [6] present rules for information hiding in specifications for Java-like languages. Their rules restrict proof obligations on method calls to only satisfy visible specifications. Consider the method call p.setX(−1) on line 32 (Figure 1). According to the supertype abstraction technique, the Point's specification must be used to reason about the correctness of such a call. The Point's specification has a protected specification case for the method setX. Thus, only privileged clients (i.e. subclasses or code in the same package) are required to obey such specifications. Since the method call on line 32 is originated from a public client (the call is located in a different package), the effective precondition on such a call defaults to **true** [6, Rule 2].

However, the instrumented code generated by current RACs ignore visibility modifiers in specifications (our second problem). Hence, by using the jmlc [4] on the same method call (line 32) results in no contract violation, but the effective precondition (assuming that the dynamic type of the receiver p is ScreenPoint) that is checked is the disjunction of the precondition on line 6 with the one in line 16 ((x >= 0) || (x < 0)), instead of the default one explained. Due to the server side instrumentation approach adopted by RACs, all specifications (with different visibility levels) are checked without respecting the information hiding rules [6]. It is important to note that none of existing tools [4] check visibility rules properly in interface specification languages. According to Leavens and Müller [6], the practical enforcement of such visibility rules is future work.

Library Checking Problem. Nowadays we have a large-scale reuse of components. This is due to the standardization of large libraries and frameworks in popular programming languages such as C++, Java, and C#. Such a standardization and heavy use of libraries keep module specification important and useful. However source code of libraries is not available for proprietary libraries [5]. This issue poses our third problem with runtime verification of OO programs. Since the contemporary RACs (e.g. jmlc [4]) need the source code in order to generate the runtime checks, we can neither specify nor verify programs during runtime when source code is not available.

2. Client-Aware Interface Specifications

To solve the three afore-mentioned problems, we propose the notion of *client-aware interface specifications*, or CAIS. We call our approach client-aware because all clients must be aware of the formal specifications contained in a special interface. We say special

interface in the sense that specifications do not necessarily be written in the source code.

Formulation of Client-Aware Interface Specifications. Figure 2 illustrate the formulation of client-aware interface specifications. For simplicity, we just consider the type Point (lines 1−7), its specifications (lines 8−15), and a client (lines 16−23). In a program logic, CAIS are embodied by the proof rule for method calls, which allows us to derive $\{P\}\ p.m()\ \{Q\}$ only from a specification (pre_m^T, $post_m^T$) associated with the static type T for the receiver p. Usually, an automated verifier uses weakest precondition semantics and achieves modularity by replacing a call $p.m()$ by the sequence of "**assert** $pre_m^T[\vec{a}/\vec{f}]$; **assume** $post_m^T[\vec{a}/\vec{f}]$" [1]. We use the notation $[\vec{a}/\vec{f}]$ to denote the substitution of the formal parameters by the actual ones. Since we are concerned with runtime verification, all the **assume** statements are checked like the **assert** ones. The instrumentation in the call site can be observed on lines 19−21 (Figure 2). Therefore we use a call site instrumentation approach in contrast to existing works [4, 11]. Another important concept of our CAIS is about abstraction. According to Liskov, we should specify the behavior, but keep it separated from implementation details [9]. This is an important concept when considering libraries specification and runtime verification.

Usefulness of Our Approach. Since our instrumentation mechanism is based on the static type of the receiver of a particular method call, we can again exploit all the benefits achieved with supertype abstraction (i.e. modular reasoning) during runtime verification. Moreover, since we adopt a static type reasoning for checking method calls, we can get similar results when using other tools like a static checker (**tackling our first problem**). As our approach is a call site driven, once our clients are known, we can instrument them according their interface specifications respecting the visibility rules (**tackling our second problem**). Finally, with client-aware interface specifications (as observed in Figure 2), one can detail specify and check during runtime the behavior of class libraries even if their source code are not available. The runtime verification is possible since our CAIS uses a client side instrumentation. Hence, we neither need the source code nor modify proprietary bytecode APIs (**tackling our third problem**).

Tool. We built these ideas on the Aspect-JML RAC compiler (ajmlc) [11] which is available online at `http://www.cin.ufpe.br/~hemr/JMLAOP/ajmlc.htm` (its current release is 3.0).

Evaluation. We intend to conduct experiments with real systems and compare the traditional runtime assertion checkers with our new approach proposed here and embedded in ajmlc [11]. So, we are looking for bugs that the contemporary runtime assertion checkers do not catch. Additionally, we want to analyze how precise is the error reporting including the visibility specifications. This is intended to blame different kinds of clients (e.g. subclasses). We also intend to evaluate the impact of our approach in relation to the classical ones in terms of source code and bytecode instrumentation sizes. Eventually, we are also interested to analyze the runtime performance of each approach.

Limitation. Since our approach is based on clients, the more new clients we have, more instrumentation code will be generated. On the other hand, in the classical approach the instrumentation is achieved only once at the declaration side which the methods being called are physically declared.

Future Work. We hope to increase expressiveness of the client-aware interface specifications to enable specifications of more complex design rules which are already found in JML [8]. We need to adapt our approach to use model program specifications [12]. Thus, this allows us to go beyond the traditional black box approach [3]. We also intend to investigate how to improve the separation of the design by contract concern in a separated interface. Preliminaries results can be found in [10].

Summary. Our hypothesis is that by using the client-aware interface specifications developers can achieve a more precise runtime verification of constrained OO programs. Our Benefits include: (i) modular reasoning by the use of supertype abstraction without drawbacks caused by runtime verification; (ii) the choice to switch from a static checker to a runtime assertion checker without surprises while getting error reporting; (iii) applying runtime verification including visibility levels achieved by information hiding principles, and (iv) precise specification and runtime verification of class libraries even if the source code is not available.

Acknowledgments

I would like to thank Professors Ricardo Lima and Gary T. Leavens (my supervisors) for the fruitful discussions we had about the ideas of my PhD thesis proposal.

References

[1] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *SIGSOFT Softw. Eng. Notes*, 31:82–87, September 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1108768.1108813.

[2] 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.

[3] M. Buchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical report, 1999.

[4] L. Burdy et al. An overview of JML tools and applications. *Int. Journal on Soft. Tools for Tech. Transfer (STTT)*, 7(3):212–232, June 2005. URL `http://dx.doi.org/10.1007/s10009-004-0167-4`.

[5] G. T. Leavens. The future of library specification. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 211–216, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0427-6. doi: http://doi.acm.org/10.1145/1882362.1882407.

[6] 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. URL `http://dx.doi.org/10.1109/ICSE.2007.44`.

[7] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995. doi: http://dx.doi.org/10.1007/BF01178658.

[8] G. T. Leavens et al. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[9] B. Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM. ISBN 0-89791-266-7. doi: http://doi.acm.org/10.1145/62138.62141.

[10] 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*. Brazilian Computer Society, 2011. to appear.

[11] H. Rebêlo et al. 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. ISBN 978-1-59593-753-7. doi: http://doi.acm.org/10.1145/1363686.1363745.

[12] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *Proceedings of the 22nd OOPSLA*, OOPSLA '07, pages 351–368, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297053.