Behavioral Contracts and Behavioral Subtyping

Robert Bruce Findler
Rice University
6100 South Main; MS 132
Houston, TX 77006; USA

Mario Latendresse
Rice University

Matthias Felleisen
Rice University
Northeastern University

ABSTRACT

Component-based software manufacturing has the potential to bring division-of-labor benefits to the world of software engineering. In order to make a market of software components viable, however, producers and consumers must agree on enforceable software contracts.

In this paper, we show how to enforce contracts if components are manufactured from class and interface hierarchies. In particular, we focus on one style of contract: pre- and post-conditions. Programmers annotate class and interface methods with pre- and post-conditions and the run-time system checks these conditions during evaluation. These contracts guarantee that methods are called properly and provide appropriate results.

In procedural languages, the use of pre- and post-condition contracts is well-established and studies have demonstrated its value. In object-oriented languages, however, assigning blame for pre- and post-condition failures poses subtle and complex problems. Specifically, assigning blame for malformed class and interface hierarchies is so difficult that none of the existing contract monitoring tools correctly assign blame for these failures. In this paper, we show how to overcome these problems in the context of Java. Our work is based on the notion of behavioral subtyping.

1. INTRODUCTION

In 1969, McIlroy [19] proposed the idea of reusable software components. In a marketplace with reusable components, software manufacturers would produce software components with well-specified interfaces. Programmers would assemble systems from these off-the-shelf components, possibly adapting some with wrapper code or adding a few new ones. If a component were to break its promises, a programmer would replace it with a different one. If a manufacturer were to improve a component, a programmer could improve the final product by replacing the link to the old component with a link to the new one.

To make such a component marketplace work, components must come with interfaces that specify their key properties. Beugnard et al [2] list four levels of component contracts:

- syntactic contracts, *e.g.* types,
- behavioral contracts, *e.g.* pre- and post-condition invariants,
- sequencing contracts, *e.g.* threading and timing constraints, and
- quality of service contracts, *e.g.* time and space guarantees.

In this paper, we focus on behavioral contracts in the form of pre- and post-conditions, for Java [10] and other object-oriented programming languages.

In principle, one could try to prove the correctness of behavioral contracts. For example, the Extended Static Checking group at Digital has developed verification tools for Java and Modula 3 [3]. In general, however, the languages used to express behavioral contracts are rich enough that it is not possible to verify statically that the contracts are never violated. In fact, the Extended Static Checking group's tools are neither complete nor sound, that is, they may validate code that is faulty and may fault code that is correct. Furthermore, most tools that do attempt to prove behavioral contracts correct are computationally expensive. Finally, these tools also require training in logic that most programmers do not possess. Because of these difficulties, we focus on tools that monitor the correctness of contracts at run-time.

Run-time enforced behavioral contracts have been studied extensively in the context of procedural languages [11, 17, 23, 25]. Rosenblum [25], in particular, makes the case for the use of assertions in C and describes the most useful classes of assertions. Adding behavioral contracts to an object-oriented language, however, poses subtle and complex problems. In particular, the contracts on overriding methods are improperly synthesized from the programmer's original contracts in all of the existing contract monitoring systems [5, 9, 12, 13, 14, 18, 21, 24]. This flaw leads to mis-assigned, delayed, or even entirely missing blame[1] for contract violations.

We overcome these flaws by basing our work on that of America [1], Liskov and Wing [15, 16], and Meyer [20] who have studied the problem of relating types and subtypes, with behavioral specifications, in an object-oriented world. Accordingly, one type is

[1] We believe that a certain amount of accountability and pride in quality craftsmanship is critical to good software production. Thus, when we use the term "blame" we mean that the programmer should be held accountable for shoddy craftsmanship.

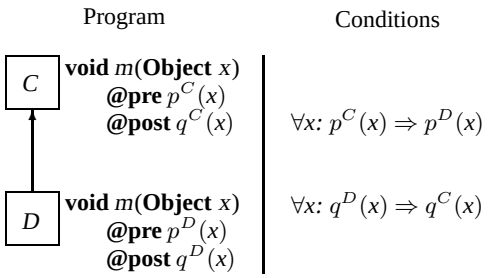

Figure 1: The Behavioral Subtyping Condition

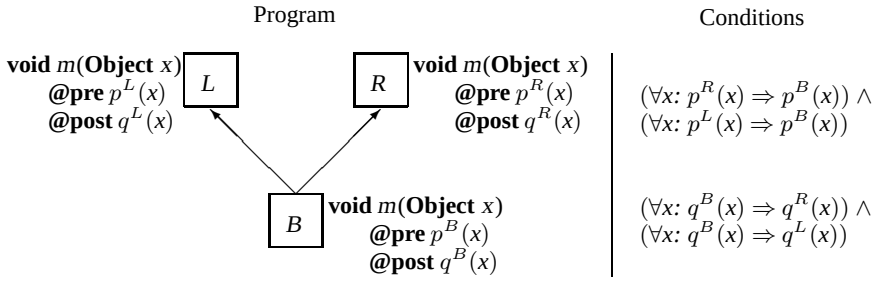

Figure 2: The Behavioral Subtyping Condition, Generalized to Multiple Inheritance

a behavioral subtype of another if objects of the subtype can be substituted in any context expecting the original type, without any effect on the program's behavior. This paper demonstrates how to integrate contracts properly, according to the notion of behavioral subtyping, into a contract monitoring tool for Java.

The next section explains behavioral subtyping in more detail. Section 3 discusses existing contract monitoring tools and shows how they fail to enforce contracts properly. Section 4 presents our contract monitoring tool. The last two sections discuss related work and draw some preliminary conclusions.

2. THE BEHAVIORAL SUBTYPING CONDITION

Behavioral subtyping [1, 15, 16, 20] guarantees that all objects of a subtype preserve all of the original type's invariants. Put differently, any object of a subtype must be *substitutable* for an object of the original type without any effect on the program's observable behavior. For pre- and post-conditions, the behavioral subtyping condition states that the pre-condition contracts for a type imply the pre-condition contracts for each subtype and the post-condition contracts for each subtype imply the post-condition contracts for the type.

Consider figure 1. It represents a program with two classes, C and D, with D derived from C. Both classes have a method m, with D's m overriding C's m. Each method, however, has its own distinct pre-condition and post-condition. Interpreted for this example, the behavioral subtyping condition states that for any input to the method, x, $p^C(x)$ implies $p^D(x)$ and $q^D(x)$ implies $q^C(x)$.

We generalize the behavioral subtyping condition to multiple inheritance by considering each subtype relationship independently. For an example, consider figure 2. It contains three interfaces, L, R, and B. Since each inheritance relationship is considered separately, we only require that B is independently substitutable for either L and R, as reflected in the conditions listed in figure 2. This is the minimum requirement to match the spirit of the behavioral subtyping condition.[2]

3. PROBLEMS WITH PRIOR WORK

We examined four tools that implement Eiffel-style [21] contracts for Java: iContract [14], JMSAssert [18], jContractor [12], and HandShake [5]. These systems enforce contracts by evaluating pre-condition expressions as methods are called and evaluating post-condition expressions as methods return. If the pre-condition fails, they blame the calling code for not establishing the proper context. If the post-condition fails, they blame the method itself for not living up to its promise.

These tools also handle programs with inheritance. With the exception of jContractor [12], they all handle inheritance in the same manner. For each overridden method, they construct a disjunction of all of the method's super pre-conditions and a conjunction of all of the method's super post-conditions. For the program in figure 1, the systems replace the condition $p^D(x)$ with

$$p^C(x) \;||\; p^D(x)$$

[2] It is possible to imagine a stronger constraint, however. One may require that L and R's conditions be equivalent. This work applies for this stricter constraint, mutatis mutandis.

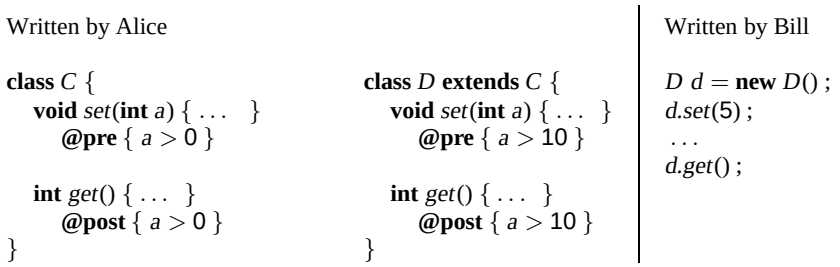

Figure 3: Delayed, Incorrect Explanation for Contract Violation

and replace the condition $q^D(x)$ with

$$q^C(x) \;\&\&\; q^D(x)$$

Since the logical statements:

$$p^C(x) \Rightarrow (p^C(x) \;||\; p^D(x))$$

and

$$(q^C(x) \;\&\&\; q^D(x)) \Rightarrow q^D(x)$$

are always true, the re-written programs always satisfy the behavioral subtyping condition, *even if the original program did not*.

As Karaorman, Hölzle, and Bruno [12, section 4.1] point out, contract monitoring tools should check the programmer's original contracts, because checking the synthesized contracts can mask programmer errors. For example, an erroneous contract formulation may never be reported. Here is their example:

```
interface I {
  int m(int a);
    @pre { a > 0 }
}
```

The interface contains a single method, *m*, with the pre-condition requirement that *a* is greater than 0. Now, imagine this extension:

```
interface J extends I {
  int m (int a);
    @pre { a > 10 }
}
```

The programmer has made a mistake in extending *I* with *J*, because *I*'s pre-condition is stronger than *J*'s pre-condition. For example, when *a* is 5, *J*'s pre-condition is false but *I*'s is true. Since *J* is a subtype of *I*, the behavioral subtyping condition tells us that the pre-condition for *I* must imply the pre-condition for *J*. This error may have been a logical error on the part of the programmer, or it may have been a typo. In either case, the tool should report the error to the programmer. If the tool does not report this error, the program may signal an incorrect error or even produce an erroneous result. To improve software's reliability, these defects must be detected and reported as soon as they occur.

None of the existing tools for monitoring pre- and post-conditions, even jContractor, handle this situation properly, in general. Instead, most combine pre-conditions with a disjunction, replacing *J*'s pre-condition with

$$(a > 0) \;||\; (a > 10)$$

which masks the defect in the program. When a programmer invokes *m* with 5, the synthesized pre-condition is true, and no error is signalled.

Similarly, an erroneous contract formulation may trigger a bad explanation of a run-time error. Figure 3 contains a program fragment that illustrates this idea. It consists of two classes, *C* and *D*, both of which implement an integer state variable. In *C*, the state variable is allowed to take on all positive values and in *D*, the state variable must be strictly larger than 10. Here, Alice wrote a hierarchy that does not match the behavioral subtyping condition, because *D* is not a behavioral subtype of *C*. Since the existing tools combine the pre-conditions with a disjunction, *D*'s pre-condition is effectively the same as *C*'s and does not guarantee that the state variable is larger than 10. Thus, the call to *set* with 5 will not signal an error. Then, when *get* is invoked, it will return 5, which incorrectly triggers a post-condition violation, blaming Alice with an error message. Even though the blame is assigned to the guilty party in this case, it is assigned after the actual violation occurs, making the problem difficult to reproduce. Also, the blame is justified with an incorrect reason, making the problem difficult to understand.

Existing contract monitoring systems handle Java's multiple inheritance in a similarly flawed manner. When a single class implements more than one interface, JMSAssert [18] collects both the pre-conditions and post-conditions together in conjunctions, ensuring that the object meets all of the interfaces simultaneously. iContract [14] collects all of the pre-conditions in a disjunction and post-conditions in a conjunction. Again, since these manufactured contracts do not match the programmer's written contracts, blame for faulty programs may be delayed, mis-assigned, or missing entirely.

4. PROPERLY MONITORING CONTRACTS

Programmers make mistakes. Their mistakes range from simple typos to complex, subtle logical errors. Accordingly, tools should

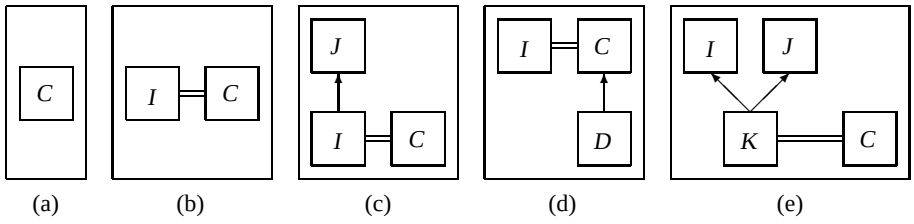

Figure 4: Section 4.2 Overview

not make the assumption that programmers have constructed well-formed programs; in particular, they should not re-write the programs based on such an assumption. Instead, tools should report errors based on the program text that the programmers provide. Giving programmers good explanations in terms of their original programs helps them pinpoint their mistakes, in a precise and timely fashion. This is especially true for contract monitoring tools, whose purpose is to provide checkable specifications of programs to improve software reliability.

4.1 Hierarchy Errors

Consider the program in figure 5. Imagine that three different programmers, Cathy, David, and Ellen, wrote the three different parts of the program. When Ellen's code invokes the static method *create_object*, it returns an instance of *C*, but with type *I*. Then, Ellen's fragment invokes *m* with 5. According to the contracts for *I*, this is perfectly valid input. According to the contract on *J*, however, this is an illegal input. The behavioral subtyping condition tells us that *J* can only be a subtype of *I* if it is substitutable for *I* in every context. When *a* is 5, *J* is *not* substitutable for *I*, so *J* is not a behavioral subtype of *I*. In short, Cathy's claim that *J* **extends** *I*, is wrong. The blame for the contractual violation must lie with Cathy.

```
// Written by Cathy
interface I {
    void m(int a) ;
        @pre { a > 0 }
}

interface J extends I {
    void m(int a) ;
        @pre { a > 10 }
}
```

```
// Written by David
class C implements J {
    static I create_object() {
        return new C() ;
    }
    void m(int a) { ... }
        @pre { a > 10 }
}
```

```
// Written by Ellen
I i = C.create_object() ;
i.m(5) ;
```

Figure 5: Hierarchy Blame

Cathy's code does not violate a pre-condition or a post-condition. Instead, the two pre-conditions have the wrong relationship. Hence, a contract checking tool should check for, and report, three different types of errors: pre-condition errors, post-condition errors, and *hierarchy extension* errors.

4.2 How to Check Contracts and Assign Blame

We model contract monitoring as a translation from Java with contracts to plain Java, augmented with three new statements: **preBlame**, **postBlame**, and **hierBlame**. Each accepts a string naming the class that is to be blamed for the respective failure. When they are executed, the program halts with an appropriate error message that blames the author of the class named by the argument.

In general, the contract compiler transfers pre-condition and post-condition contracts into wrapper methods[3] that check the contracts and call the corresponding original method. It rewrites calls to methods with contracts into calls to the appropriate wrapper method. Furthermore, method calls in the elaborated program are rewritten to call these wrapper methods, based on the static type of the object whose method is invoked. The translation thus depends on the type analysis and takes into account the type hierarchy.

Figure 4 shows a series of hierarchy diagrams that provide an outline for this section. Each diagram corresponds to a configuration of classes and interfaces. The boxes represent classes and interfaces. The classes are named *C* and *D* and the interfaces are named *I*, *J*, and *K*. The single lines with arrow-heads represent both class and interface inheritance and the double lines without arrow-heads represent interface implementation.

Diagram 4 (a) illustrates the simplest case. Figure 6 contains program text corresponding to this diagram and its translation. The program consists of two classes, *C* and *Main*. The original *C* class has a method *m* with a pre-condition and a post-condition. Its translation has two methods, the original *m* and the wrapper method *m_C*. The name of the wrapper method is synthesized from the name of the original method and the name of the class. The wrapper method accepts one additional argument naming the class that is calling the method, which is blamed if the pre-condition fails. In figure 6 lines 4–6, the wrapper method checks the pre-condition and blames the class of the caller if a violation occurs. Then, in line 7, it runs the original method. Finally, in lines 8–10, it checks the post-condition, blaming the class itself for any violations of the post-condition. The contract compiler also rewrites the call to *m* in *Main* to call the wrapper method, passing in `"Main"` to be blamed for a pre-condition violation.

[3] Although it may be possible to avoid adding wrapper methods, we believe wrapper methods are the simplest and most efficient approach to proper contract checking.

```
class C {
  void m(int a) { ... }
    @pre
      ... C's pre-condition ...
    @post
      ... C's post-condition ...
}

class Main {
  public static void
  main(String[] argv) {
    new C().m(5) ;
  }}
```

```
1:  class C {
2:    void m (int a) { ... }
3:    void m_C (string tbb, int a) {
4:      if (! ... C's pre-condition ... ) {
5:        preBlame(tbb) ;
6:      }
7:      m(a) ;
8:      if (! ... C's post-condition ... ) {
9:        postBlame("C") ;
10:     }}}

12: class Main {
13:   public static void
14:   main(String[] argv) {
15:     new C().m_C("Main", 5) ;
16:   }}
```

Figure 6: Pre- and Post-condition Checking

Diagram 4 (b) contains a class and an interface. The class implements the interface. As with the previous example, when a method is called, its pre-condition must be checked, though the pre-condition to be checked depends on the static type of the reference to the object whose method is invoked. Since that may be either I or C, two wrapper methods are generated: *m_C* and *m_I*, which each check their respective pre- and post-conditions.

The example in diagram (b) adds another twist. Since instances of C are substitutable in contexts expecting Is, we must also check that the hierarchy is well-formed. In this case, I's pre-conditions must imply C's pre-conditions and C's post-conditions must imply I's post-conditions, for each method call to m. There are four possibilities for C and I's pre-conditions. Clearly, if both are true, no violation has occurred and if both are false, the pre-condition does not hold and the caller must be blamed. If I's pre-condition is true and C's pre-condition is false, the hierarchy is malformed and the author of C must be blamed. If I's pre-condition is false and C's pre-condition is true, the hierarchy is well-formed and no hierarchy violation is signaled. In this case, however, if the object is being viewed as an instance of I, the pre-condition checking code in *m_I* blames the caller for failing to establish the pre-condition. If the object is being viewed as an instance of C, no error occurs and no violation is signaled. The logic of post-condition checking is similar.

To perform the hierarchy checks, hierarchy checking methods are generated for each interface and class method. For classes, the new methods are inserted into the translated version of the class. For interfaces, the new methods are inserted into a new class that is generated for each interface. These hierarchy checking methods recursively combine the result of each pre- or post-condition with the rest of the pre- and post-condition results in the hierarchy to determine if the hierarchy is well-formed.

Figure 7 contains a translation that illustrates how our compiler deals with diagram (b). The wrapper methods, *m_C* and *m_I*, are augmented with calls to the hierarchy checking methods, *m_pre_hier* and *m_post_hier* in figure 7 lines 11 and 16. The *m_pre_hier* and *m_post_hier* methods in C ensure that the pre- and post-condition hierarchies are well-formed. The checkers for I would appear in the *I_checkers* class; they are analogous and omitted.

For the pre-condition checking *m_pre_hier* accepts the same arguments as the original method and returns the value of the pre-condition. To check the hierarchy, the method first calls *I_checkers*'s *m_pre_hier* method in line 20, which ensures that the pre-condition hierarchy from I (and up) is well formed. Since *this* in *I_checkers* does not refer to the object whose contracts are checked, the current object is passed along to *I_checkers*'s *m_pre_hier*. In our example, the hierarchy from I (and up) is trivially well-formed, since I has no supertypes. The result of *I_checkers*'s *m_pre_hier* is the value of I's pre-condition on m and is bound to *sup* in C's *m_pre_hier*, as shown on line 19. Then, *m_pre_hier* binds *res* to the value of its own pre-condition, in line 21. Next, it tests if I's pre-condition implies C's pre-condition, with the expression *!sup || res* in line 22, which is logically equivalent to $sup \Rightarrow res$. If the implication holds, *m_pre_hier* returns the result of the pre-condition, in line 23. If not, it evaluates the **hierBlame** statement in line 25, which aborts the program and blames C as a bad extension of I.

The post-condition checking recursively traverses the interface and class hierarchy in the same order as pre-condition checking. In contrast to the pre-condition checking, post-condition checking accumulates the intermediate results needed to check the hierarchy instead of returning them. In our example, the first two arguments to *m_post_hier* in C are the accumulators: *tbb* (figure 7 line 27) is the class to be blamed for the failure and *last* is the value of the post-condition of a subtype (initially false if there are no subtypes). To determine if there is a hierarchy violation, *res* is bound to the value of m's post-condition in line 28, and the implication is checked in line 29. If the hierarchy is flawed at this point, *tbb* is blamed in line 32. In this example, this cannot happen, since *res* is initially false, but the code is needed in general. Then, *I_checkers*'s *m_post_hier* is called in line 30, with the value of C's post-condition and C's name. Thus, the blame for a bad hierarchy discovered during *I_checkers*'s *m_post_hier* falls on C.

Diagram 4 (c) adds interface checking to the picture. Still, the contract checkers for the program in diagram (c) are similar to those in diagram (b). The additional interface generates an additional class for checking the additional level in the hierarchy.

```
interface I {
   void m(int a) ;
      @pre
        ... C's pre-condition ...
      @post
        ... C's post-condition ...
}

class C implements I {
   void m(int a) { ... }
      @pre
        ... I's pre-condition ...
      @post
        ... I's post-condition ...
}
```

```
 1: interface I { ... }
 2: class I_checkers { ... }

 4: class C implements I {
 5:    void m () { ... }
 6:    void m_I (string tbb, int a) { ... }
 7:    void m_C (string tbb, int a) {
 8:       if (! ... C's pre-condition ... ) {
 9:          preBlame(tbb) ;
10:       }
11:       m_pre_hier(a) ;
12:       m(a) ;
13:       if (! ... C's post-condition ... ) {
14:          postBlame("C") ;
15:       }
16:       m_post_hier("C", false, a) ;
17:    }
18:    boolean m_pre_hier(int a) {
19:       boolean sup =
20:          I_checkers.m_pre_hier(this, a) ;
21:       boolean res = ... C's pre-condition ;
22:       if (!sup || res) {          // sup ⇒ res
23:          return res ;
24:       } else {
25:         hierBlame("C") ;
26:       }}
27:    void m_post_hier(string tbb, boolean last, int a) {
28:       boolean res = ... C's post-condition ;
29:       if (!last || res) {          // last ⇒ res
30:          I_checkers.m_post_hier ("C", res, this, a) ;
31:       } else {
32:         hierBlame(tbb) ;
33:       }}}
```

Figure 7: Hierarchy Checking

Diagram 4 (d) introduces class inheritance (or implementation inheritance) which poses a more complex problem for our compiler. As with an additional interface, new methods are generated to check the hierarchy. Unlike an additional interface, the new hierarchy checking methods are only used when an instance of the derived class is created. That is, if the program only creates instances of C, the hierarchy below C is not checked. Instances of D, however, do check the entire hierarchy, including C's and I's pre- and post-conditions. In general, the conditions of every interface and every superclass of the originally instantiated class are checked at each method call and each method return to ensure the hierarchy is sound.

Diagram 4 (e) shows an interface with two super-interfaces. According to the discussion in section 2, the hierarchy checkers must check that the pre-condition in I implies the pre-condition in K and the pre-condition in J implies the pre-condition in K.[4] The following boolean identity

$$(a \rightarrow c) \wedge (b \rightarrow c) \Leftrightarrow (a \vee b) \rightarrow c$$

tells us that we can just check that disjunction of I's and J's pre-conditions implies K's pre-condition. Accordingly, as shown in figure 8, *K_checkers*'s *m_pre_hier* method hierarchy checker combines the results of *I_checkers*'s and *J_checkers*'s *m_pre_hier* methods in a disjunction and binds that to *sup* in figure 8 lines 6–8. Thus, the contract checker's traversal of the type hierarchy remains the same.

For post-conditions, we take advantage of a similar boolean identity:

$$(a \rightarrow b) \wedge (a \rightarrow c) \Leftrightarrow a \rightarrow (b \wedge c)$$

and combine the recursive calls with a conjunction to compute the result of the post-condition hierarchy checking method, as shown in *m_post_hier*'s definition in figure 8 lines 19–22.

4.3 Implementation

An implementation of our contract checker is in progress. The second author, Mario Latendresse, is building a Java contract compiler. We plan to release a prototype by the end of 2001.

To interact with other Java tools, interfaces should compile to a single `.class` file. As described here, the contract compiler generates an additional class for each interface. Our implementation,

[4] Another alternative, as mentioned in footnote 2, is to ensure that I's and J's conditions are equivalent. This could easily be checked at this point in the hierarchy checker.

```
interface I {
   void m(int a);
      @pre
       ... I's pre-condition ...
      @post
       ... I's post-condition ...
}


interface J {
   void m(int a);
      @pre
       ... J's pre-condition ...
      @post
       ... J's post-condition ...
}


interface K extends I, J {
   void m(int a);
      @pre
       ... K's pre-condition ...
      @post
       ... K's post-condition ...
}
```

```
 1: class I_checkers { ... }
 2: class J_checkers { ... }

 4: class K_checkers {
 5:   static boolean m_pre_hier(K this, int a) {
 6:     boolean sup =
 7:         I_checkers.m_pre_hier(this, a) ||
 8:         J_checkers.m_pre_hier(this, a) ;
 9:     boolean res = ... K's pre-condition ... ;
10:     if (!sup || res) {          // sup ⇒ res
11:       return res ;
12:     } else {
13:       hierBlame("K") ;
14:     }}
15:   static void m_post_hier(string tbb, boolean last,
16:                           K this, int a) {
17:     boolean res = ... K's post-condition ... ;
18:     if (!last || res) {         // last ⇒ res
19:       return
20:         I_checkers.m_post_hier("K", res, this, a)
21:         &&
22:         J_checkers.m_post_hier("K", res, this, a);
23:     } else {
24:       hierBlame(tbb) ;
25:     }}}
```

Figure 8: Hierarchy Checking for Multiple Inheritance

however, augments the `.class` file generated for the interface with enough information to add the wrapper and hierarchy methods to each class that implements the interface. This is done using a custom attribute in the class file that contains the byte-codes of the contracts. The hierarchy checking methods for interfaces are then copied into classes that implement interfaces.

In the code examples in section 4.2, we used method names for wrappers that are valid Java identifiers. In our implementation, special names for wrapper methods are used in the class files to eliminate name clashes with programmer-defined method names. Additionally, our contract compiler does not add new blame-assigning statements to Java; instead it inlines code that raises an exception to blames the guilty party.

Our contract compiler does not install a class loader, it does not generate any new `.java` or `.class` files, nor does it require any existing class libraries during evaluation. These features of our design enable our contract compiler to integrate seamlessly with the existing Java development environments, unlike existing Java contract checkers.

Since our contract compiler uses wrapper methods to check contracts and it redirects each method call to call the wrapper methods, the programmer's original methods are still available in the class. Thus, the `.class` files that our contract compiler generates can be linked to existing, pre-compiled byte-codes. This allows pre-existing, byte-code distributions of Java code to interoperate with code compiled by our contract compiler. The pre- and post-conditions of methods invoked by the pre-existing bytes-codes are not checked.

5. RELATED WORK

Karaorman, Hölzle, and Bruno [12, section 4.1] first recognized the problems with re-writing the programmer's pre- and post-conditions. Although they recognize the problem, their contract checking tool still does not check the hierarchy properly.

In addition to the already mentioned contract monitoring tools, much pre- and post-condition based work in object-oriented languages would benefit from considering hierarchy violations in addition to pre- and post-condition violations.

Edwards et al's [6] paper on detecting interface violation in component software provides a mechanism for separating contracts from the components that implement them. They map each object to a parallel "mathematical" representation of the object and check contracts on the parallel objects. This technique neatly sidesteps problems when contracts operate on the original objects, such as contracts that mutate objects, but there is no guarantee that, when the parallel objects satisfy the contracts, the original objects also satisfy the contracts.

Dhara and Leavens [4] describe a system for proving pre- and post-condition style assertions on methods. Like the tools in section 3, their tool combines pre-conditions with a disjunction and post-conditions with a conjunction. Accordingly, they suffer from the same problems as the other tools surveyed in section 3. The techniques used here should easily generalize to their work.

6. CONCLUSIONS AND FUTURE WORK

This paper argues that contract-monitoring tools should report three kinds of errors: pre-condition errors, post-condition errors, and hierarchy violations. It demonstrates that, without the last kind of

error, existing contract tools assign blame incorrectly for certain contractual violations, sometimes do not catch certain contractual violations at all, and provide bad explanations for programmer's errors in other cases. The paper also explains how to implement a contract checker that properly checks for hierarchy violations.

This work demands a complete semantic characterization of contracts and contract monitoring, based on a rigorous semantic description of the programming language. A companion paper [7] treats contracts as an extension of types and presents a contract soundness theorem, akin to the type soundness of Milner [22].

Java suffers from a flaw that our contract checker exacerbates. As America [1] and Szyperski [26] point out, implementation inheritance and interface inheritance are two separate mechanisms, yet Java combines them. In practice, this combination means that programmers may extend a class without intending the derived class to be a subtype and thus, not a behavioral subtype. Unfortunately, our contract checker may still assign blame to the derived class for not being a behavioral subtype. The cleanest solution to this problem, in Java, is to use the proxy pattern [8], but this is not always practical. We are considering an extension to the translator presented here that allows programmers to explicitly specify that a derived class is not a behavioral subtype.

7. ACKNOWLEDGEMENTS

Thanks to Daniel Jackson, Shriram Krishnamurthi, and Clemens Szyperski for valuable comments on drafts of this paper.

8. REFERENCES

[1] America, P. Designing an object-oriented programming language with behavioural subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, Lecture Notes in Computer Science, pages 60–90. Springer-Verlag, 1991.

[2] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, june 1999.

[3] Detlefs, D. L., K. Rustan, M. Leino, G. Nelson and J. B. Saxe. Extended static checking. Technical Report 158, Compaq SRC Research Report, 1998.

[4] Dhara, K. K. and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings 18th International Conference on Software Engineering*, pages 258–267. IEEE, 1996. Berlin, Germany.

[5] Duncan, A. and U. Hölze. Adding contracts to Java with handshake. Technical Report TRCS98-32, The University of California at Santa Barbara, December 1998.

[6] Edwards, S., G. Shakir, M. Sitaraman, B. Weide and J. Hollingsworth. A framework for detecting interface violations in component-based software,. In *Proceedings 5th International Conference on Software Reuse*, pages 46–55. IEEE, June 1998.

[7] Findler, R. B. and M. Felleisen. Contract soundness for object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.

[8] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[9] Gomes, B., D. Stoutamire, B. Vaysman and H. Klawitter. *A Language Manual for Sather 1.1*, August 1996.

[10] Gosling, J., B. Joy and G. Steele. *The Java(tm) Language Specification*. Addison-Wesley, 1996.

[11] Holt, R. C. and J. R. Cordy. The Turing programming language. In *Communications of the ACM*, volume 31, pages 1310–1423, December 1988.

[12] Karaorman, M., U. Hölzle and J. Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of Meta-Level Architectures and Reflection*, volume 1616 of *lncs*, July 1999.

[13] Kölling, M. and J. Rosenberg. *Blue: Language Specification, version 0.94*, 1997.

[14] Kramer, R. iContract — the Java design by contract tool. In *Technology of Object-Oriented Languages and Systems*, 1998.

[15] Liskov, B. H. and J. Wing. Behavioral subtyping using invariants and constraints. Technical Report CMU CS-99-156, School of Computer Science, Carnegie Mellon University, July 1999.

[16] Liskov, B. H. and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[17] Luckham, D. C. and F. von Henke. An overview of Anna, a specification language for Ada. In *IEEE Software*, volume 2, pages 9–23, March 1985.

[18] Man Machine Systems. Design by contract for Java using jmsassert. `http://www.mmsindia.com/DBCForJava.html`, 2000.

[19] McIlroy, M. D. Mass produced software components. In Naur, P. and B. Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138–150, 1968.

[20] Meyer, B. *Object-oriented Software Construction*. Prentice Hall, 1988.

[21] Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.

[22] Milner, R. A theory of type polymorphism in programming. *Journal of Computer Systems Science*, 17:348–375, 1978.

[23] Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[24] Plösch, R. and J. Pichler. Contracts: From analysis to C++ implementation. In *Technology of Object-Oriented Languages and Systems*, pages 248–257, 1999.

[25] Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.

[26] Szyperski, C. *Component Software*. Addison-Wesley, 1998.