Object-oriented Programming Languages Need Well-founded Contracts

Robert Bruce Findler, Mario Latendresse, Matthias Felleisen

Department of Computer Science,
6100 South Main Street
Rice University MS 132
Houston, TX 77005
USA

Rice University Computer Science Technical Report TR01-372

Abstract. Over the past few years, the notion of building software from components has become popular again. The goal is to produce systems by adapting and linking off-the-shelf modules from a pool of interchangeable components. To turn this idea into reality, the formal descriptions of software components need to specify more than the type signatures of their exported services. At a minimum, they should contain assertions about critical properties of a component's behavior. By monitoring such behavioral contracts at run-time, language implementations can pinpoint faulty components, and programmers can replace them with different ones.

In this paper, we study the notion of behavioral contracts in an object-oriented setting. While the use of behavioral contracts is well-understood in the world of procedural languages, their addition to object-oriented programming languages poses remarkably subtle problems. All existing contract enforcement tools for Java fail to catch flaws in contracts or blame the wrong component for contractual violations. The failures point to a lack of foundational research on behavioral contracts in the OOP world.

1 Components, Contracts, and Classes

In 1969, McIlroy [16] proposed the idea of reusable software components. In a market place with reusable components, software manufacturers produce interchangeable software components with well-specified interfaces. Programmers assemble systems from these off-the-shelf components, possibly adapting some with wrapper code or adding a few new ones. When a component fails to live up to its promises, a programmer replaces it with a different one. When a manufacturer improves a component, the programmer can improve the final product by replacing the link to the old component with a link to the new one.

To make this component market work, component interfaces should state more than the type signatures of exported and imported items. Ideally, interfaces should specify the entire behavior of a component; realistically, interfaces should at least specify essential aspects of the behavior. The earliest suggestion along those lines came from Parnas [20], who wanted correctness assertions in module interfaces. More recently, Beugnard, Jézéquel, Plouzeau, and Watkins [2] proposed that components should come with four levels of interface descriptions:

1. syntactic aspects (type systems, i-o dependencies);
2. behavioral guarantees (weak or strong logical assertions);
3. concurrency guarantees and warnings;
4. quality of service statements.

They refer to such rich interfaces as *contracts*. This paper focuses on *behavioral contracts*.

The usefulness of contracts depends on their enforcement. Unless contracts are verified or monitored, there is no assurance that the software works properly. Hence, it is paramount that contracts have a precise meaning and that the language implementation enforces the meaning of contracts. In particular, contract violations must be pinpointed as precisely as possible, because blaming the wrong component manufacturer (a person, a department, or an external supplier) or providing the wrong explanation for a failure may cause economic damage.

Object-oriented programming languages (OOPL) have strongly embraced the notion of components and behavioral contracts. For example, Meyer's [17] Eiffel explicitly incorporates the notion of "programming by contract." More

precisely, an Eiffel class plays the role of a component. A programmer can express behavioral contracts via logical assertions about the calling context, the return of a method, and as invariants across calls. Eiffel supports these contracts with a run-time monitoring system. If a pre-condition fails, Eiffel blames the caller of the method; if a post-condition fails, it assigns the blame to the method. Recently, several research teams have adapted Eiffel's notion of "programming by contract" to Java [5, 8, 10, 15].

Unfortunately, unlike contract monitoring tools for procedural languages, none of those for object-oriented languages do not live up to their promises. As it turns out, catching contract violations and assigning blame properly poses subtle problems. More concretely, we believe that the problems of existing tools are due to a lack of foundational work on contracts and contract violations. There is little work on the semantics of behavioral contracts, contract monitoring, and contract violations. In this paper, we present the idea of behavioral contracts for object-oriented languages in general and for Java in particular. Although we study this problem in the context of Java, the issues that we address are common to any component mechanism that includes objects. Then we discuss how the existing contract enforcement tools fail to deal with contracts properly. Finally, we formulate a challenge for the foundational software engineering community, namely, the design of a semantics of contracts and contract violations for behavioral contracts.

In the next section we briefly introduce four Java contract tools. In section 3 we look at concrete examples for which the tools fail and analyze why. In section 4, we formulate the contract enforcement problem and discuss several options concerning its solution. Section 5 discusses related work and the last section is a brief summary.

2 Behavioral Contracts in the OOP World

In Java, it is natural to argue that behavioral contracts should be associated with Java interfaces. Following Eiffel's early example [17], contracts should be logical assertions—boolean expressions involving the program's variables—about methods and classes. More concretely, a programmer can specify *pre-conditions* and *post-conditions* on a method.[1]

Also following Eiffel's example, the run-time system should monitor the logical assertions about classes and methods. In particular, it should signal the failure of a pre-condition or post-condition. In Eiffel's case, the system blames the caller of a method when a pre-condition fails, and it blames the method itself when the post-condition fails.

Useful contracts are not necessarily complete descriptions of a method's behavior. Requiring completeness would demand much more logic training for programmers than currently available. Instead, useful contracts often specify just some essential aspects of a method's behavior. Consider the following example:

```
... String get_file_name();
// The method interactively queries the user for the
// name of an existing file.
// @post: System.file_exists(@ret)
```

The informal constraint expresses the complete behavior of the method in English. The quasi-formal post-condition specifies that the result of the method is a string for which the *file_exists* predicate succeeds.

Here is a second example:

```
... void click_button(Button btn);
// The method simulates a user click on the button btn
// It is only feasible to click on a button when the button's
// top-level window is visible and has the focus.
// @pre: { Window w = System.get_top_level_focus_window();
//         w.contains(btn)
//         && btn.is_enabled()
//         && btn.is_shown() }
```

[1] Eiffel also allows programmers to specify invariants. We ignore invariants here, because dealing with pre- and post-conditions poses a sufficiently rich set of problems.

Again, the informal comment specifies the approximate behavior of the method, which is taken from a test suite for GUI programs. The quasi-formal comment expresses that it makes only sense to call this method if the button belongs to the window with the focus and has certain other properties.

Both of the examples have the property that their truth value depends on the state of the world, including the user's interactions with the computer. Other examples from our work experience include predicates that depend on extensions of a class hierarchy framework or on interactions between objects that act like higher-order functions. The general goal is to state weak but important assertions and to monitor them at run-time. Useful weak assertions are often (non-type) predicates on a method's argument, a relation among a method's arguments, or a relation among methods of a class. Still, even such weak assertions help programmers comprehend, test, and debug methods during development and maintenance. Others have confirmed this claim in the context of procedural languages [1, 9, 21].

3 Existing Contract Monitoring Tools for Java

A literature and Web search produced four tools for monitoring behavioral contracts in Java: JMSAssert [15], iContract [10], jContractor [8], and HandShake [5]. The first two are implemented and are distributed over the Web. We were not able to locate working versions of jContractor and HandShake. The authors either did not respond to our inquiries or told us that no implementation was available. We base our evaluation of these two systems on the paper designs.

The following subsection provides a brief technical survey of these tools. The second subsection shows how the tools deal with certain situations.

3.1 The Tools

All existing Java contract tools—JMSAssert [15], iContract [10], HandShake [5], jContractor [8]—work according to approximately the same principles. Both pre- and post-conditions are specified as boolean expressions for methods in classes or in interfaces. Given a program's interface and class hierarchy with contracts, the tools propagate the contracts along the inheritance hierarchy. More specifically, the tools build a disjunction of all pre-conditions and a conjunction of all post-conditions.

Once the complete pre-conditions and post-conditions have been assembled, the tools translate assertions from interfaces into run-time checks at the entrance and exit of methods.[2] Furthermore, during this step, they combine the assertions from all the interfaces that a class implements. Finally, the tools add code that deals with contract violations during execution. The code raises a Java exception whenever a condition fails, signalling a breach of contract. They also use the JVM traceback facility to show some basic status information, possibly enriched with some additional messages. It is then up to the programmer to discern the source of the problem and the faulty component.

The differences between the tools concern partly the syntactic conventions and partly the implementation of condition monitoring. For completeness we summarize the salient aspects of each design.

In JMSAssert, pre-conditions and post-conditions are specified in classes as Javadoc comments where the assertions use a special language called JMScript. A contract can be specified in an interface, in which case it is propagated to all classes that implement the interface. A preprocessor generates instrumented Java programs that can be compiled by a regular Java compiler. The JVM machine running the instrumented Java programs must use a co-interpreter for JMScript.

In jContractor, the implementor specifies the assertions as separate boolean-valued methods. These methods must be named according to a predefined convention. At run time, the tool uses a special class loader to load the proper class files dynamically, if contracts are active.

In iContract, pre-conditions and post-conditions are specified as comments above the method signature. The expressions are formulated in a superset of Java. The iContract tool uses a preprocessor to generate instrumented Java programs for every method having assertions. The resulting instrumented programs can be compiled by any Java compiler for execution by the JVM.

In HandShake, the pre-conditions and post-conditions are specified in a separate file as Java expressions. These files are compiled to a compact binary format. At run time, file system calls made by the JVM are redirected and in the case of an access to a class file with contracts, the bytecode of the assertions is added.

[2] Some designs also suggest using Java's reflection facilities to implement contract enforcement.

3.2 Experiences with the Tools

Type contracts are useful because they discover (potential) errors. For example, if a programmer specifies that some operation consumes integers and if this operation is used in an application where the argument expressions provably produce booleans, then the program is erroneous and should not be evaluated. Similarly, when a programmer specifies contracts for methods and the contracts are inconsistent, a contract monitoring tool should warn the programmer when witnesses to the inconsistency occur. Conversely, a software tool should not create spurious or unwarranted reports. In terms of our problem, a piece of code that is unaware of some contract should never be blamed for failing to respect the conditions of the contract. We will demonstrate with some examples in this section that the existing tools do not respect these basic guidelines.[3]

The first example concerns multiple interface implementation. Java explicitly allows a class to implement many interfaces. Other OOPLs also support this capability. Thus, we can have the following situation:

interface *I1* { ... } // package P1
⋮
interface *I2* { ... } // package P2
⋮

---

package *P3*;
import *P1*;
import *P2*;
class *A* **implements** *I1*, *I2* { ... }

A client of package *P3* may perceive an *A* object from a single perspective, say, *I1*. Furthermore, due to the module scoping mechanism the client does not necessarily know about all interfaces. As a matter of fact, due to Java's scoping mechanism, the client may not even have access to *I2*. Hence, the contractual annotations in *I2* should not play a role for the client. The existing tools, however, will always enforce the contracts in both *I1* and *I2*. As a result, they may blame a client of *A* for not establishing a contractual obligation from interface *I2*, even though the client does not even know about this interface.

A concrete example is presented in figure 1. Class *A* implements the two interfaces *I1* and *I2*, but only interface *I2* has a pre-condition on *x*. Class *Main* creates an *A* object and types it as *I1*. Since *I1* has no pre-condition there should not be any blame assigned to the call *a.m*(2) in *Client1*. The tool iContract, however, reports a violation of the pre-condition $x > 10$. Clearly this is due to the implementation of the instrumented class having the pre-conditions of the two interfaces at the same time. The programmer of class *Client1* should not be blamed by the breach of contract since interface *I1* has no such pre-condition.

Another major problem for assigning blame is due to *object substitutability*, that is, the ability of an instance of one class to play the role of a superclass (supertype). All these forms of substitutability pose problems in the presence of interface-based contracts. Moreover, method overriding adds the problem of modification of contracts.

If a class implements several interfaces via interface inheritance, client code may explicitly or implicitly cast an object from one interface to another. Consider the following situation:

interface *I3* { ... }

interface *I4* **extends** *I3* { ... }

class *B* **implements** *I4* { ... }

A client may now pass a *B* object of type *I4* to a method that expects an object of type *I3*. This is valid according to Java's type system—except that the object may now have different contractual obligations to satisfy.

Figure 2 presents a concrete case. The interface *PosCounter* extends with a contract the interface *Counter*. Class *Pc* implements the interface *PosCounter*. The implementor of class *Client1* uses the interface *Counter* but does not

[3] The tools also suffer from another, practically important problem, namely, that the tools report all errors in terms of instrumented code rather than source code. As a result, it is difficult, if not impossible, in many cases to find the true source of an error. We believe that a source-correlating elaborating front-end along the lines of McMicMac [11, 12] should solve this problem.

```
interface I1 {

    int m(int x);
}

class Client1 {
    public static void F(I1 a) {
        a.m(2);
    }
}
```

```
interface I2 {
    // @pre x > 10
    int m (int x);
}

class A implements I1, I2 {
    public int m (int x) { ... }
}

class Main {
    public static void main(String argv[]) {
        I1 a = new A();
        Client1(a);
    }
}
```

Fig. 1. Incorrect blame due to inconsistent contracts in multiple interfaces

know anything about the interface *PosCounter*. The implementor of class *Client2* uses interface *PosCounter* assuming its contract. But as we will shortly show, the implicit cast due to the assignment statement *cl.c = co* is problematic.

The method *Main.main* uses both clients, first by creating a positive counter *c* and passing it to *Client2.m* to be used by *Client1.m* through *cl*. The call *cl.m*() attempts to decrease the counter *c* to -1, violating *PCounter.dec*'s pre-condition. The class *Client1* should not be blamed for the breach of contract, since it knows nothing of the contract of *PCounter*. The tools, however, blame *Client1*, because *PCounter* objects always check the post-condition from all implemented interfaces.

```
interface Counter {
  int getValue();
  void inc();
  void dec();
}

interface PCounter extends Counter {
  // @pre getValue() > 0
  void dec();
}

class Pc implements PCounter {
  int val = 0;

  public int getValue(){ return val; }
  public void inc(){ ++val; }
  public void dec(){ --val; }
}
```

```
class Client1 {
  Counter c;
  void m(){ c.dec(); }
}

class Client2 {
  void m(Client1 cl, PCounter co) {
    cl.c = co;
  }
}

class Main {
  static public void main() {
    PCounter c = new Pc();
    Client1 cl = new Client1();

    new Client2().m(cl,c);
    cl.m();
  }
}
```

Fig. 2. Incorrect blame due to implicit upward casts

The last example shows how, on occasion, the tools fail to catch a problem with the programmer's contracts. Let's reconsider interface inheritance:

interface *I3* { ... }

interface *I4* **extends** *I3* { ... }

When an interface is extended, the contracts for an overridden method may be different from the original method's contract. For example, consider the program in figure 3. Method *m* in class *A* assumes that its argument is always greater than 0. Similarly, method *m* in class *B* assumes that its argument is always greater than 10. The print statements in the two methods are based on these assumptions. Method *main*, however, applies *B*'s *m* to 5, and the method program produces this output:

```
this never prints a number smaller than 10: 5
```

This is the fault of the tools. There are two possibilities: either the contract is bad, in which case the tools should report that the contract is bad, or the contract is not bad, in which case the tools should enforce the contract.

The tools surveyed, however, combine pre-condition contracts in a disjunction so they do not discover that the clause from *J* evaluates to false when the argument is 5. Since we believe that tools should not assume that the programmer's contracts are correct and that they should instead uncover potential problems, we claim that this kind of oversight is a major short-coming of the tools.

4 The Challenge

The failure of existing contract monitoring tools reflects a lack of foundational work in software engineering. If we wish to formulate contracts in component interfaces, we should formulate *a semantics of contracts* and, in particular, *a semantics of contract violations*. For the latter aspect, the semantics should include a connection between the violation and the code that caused the violation.

A closer look at the examples in section 3.2 shows that they break the Liskov-Wing principle of object substitutability. Roughly speaking, the principle says that an object of type *T* can play the role of a supertype object if it behaves correctly in all contexts where a supertype object is legal. In particular, a supertype pre-condition should imply a subtype pre-condition and a subtype post-condition should imply a supertype post-condition. Furthermore, if a class has several interfaces the contracts for shared methods should be equivalent.

While the contracts of section 3.2 fail to satisfy these conditions, the true question is how to test the contracts at run-time and how to assign blame in case the conditions fail. Further reflection suggests two interpretations. The first, "conservative" interpretation requires that contracts in class and interface hierarchies are always formulated according to the Liskov-Wing principle. If a monitoring tool discovers at run-time that the contracts are violated, it may have to blame the hierarchy itself. The second, "liberal" interpretation requires only that no bad object substitutions occur. That is, it accepts the class hierarchy as a notational convenience for writing down contracts. Inheritance only takes place if an object of interface *I* is substituted into a context of type *J* and *I* is an extension of *J*. If in addition the object now violates the Liskov-Wing implications between *I* and *J*, a contract violation occurred. The following two subsections present these two interpretations in more detail and discuss how tools could assign blame in these contexts.

4.1 Assigning Blame Assuming Semantic Substitutability

One way to interpret the tools' behavior is to say that they attempt to implement the Liskov-Wing principle but make the mistaken assumption that *programmers don't write down meaningless contract extensions*. Put differently, the tools assume that programmers always wish to add assertions to a pre-condition of a method in a disjunctive manner and that they always wish to add assertions to the post-condition in a conjunctive manner. As figure 3 shows, however, this assumption is flawed, because programmers may easily add flawed statements. In this particular example, the programmer may have wished to write -10 instead of 10. As a result, the monitoring tools will not discover when an interface extension places inconsistent pre-conditions or post-conditions on an overridden method. Since weak tools such as monitoring tools should attempt to find potential problems (rather than prove programs correct), we believe the tools are flawed.

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

class A implements I {
  public void m(int a) {
    System.out.println("this never prints a number smaller than 0: " + a);
  }
}

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

class B extends A implements J {
  public void m(int a) {
    System.out.println("this never prints a number smaller than 10: " + a);
  }
}

class Main {
  public static void main(String argv[]) {

  I a = new A();
  J b = new B();

  a.m(5);
  b.m(5);
  }
}
```

Fig. 3. Failure to blame, due to bad contract assembly

To remedy the situation, the tools should test the proper implications at run-time instead of disjunctions and conjunctions. Then, if one of the implications fails at run-time, the tools should blame the caller, the callee, or the interface and class hierarchy. Consider the hierarchy of pre-conditions for example. If a pre-condition fails for a given execution state, the tools should trace the hierarchy in an upward fashion, starting from the current method. If an overridden method is found, its pre-condition is evaluated with the same values. Now, if this pre-condition is true, the hierarchy is faulty, because it does not satisfy the Liskov-Wing sub-typing rules. In contrast, if this pre-condition is also false and it is the overridden method from the class where the calls originate, the caller is blamed. For all other methods, the search continues. Post-conditions are treated in an analogous fashion, but the implications are inverted.

For example, the failure of the call *cl.m*(), in Figure 2, would initiate a traceback to the method *Counter.dec*. The absence of a pre-condition makes it true, resulting in a blame to the hierarchy.

Adapting this contract monitoring semantics for all of Java requires two extensions of the Liskov-Wing work. First, the work requires an interpretation of classes that implement more than one interface. Second, the work requires an interpretation of multiple interface inheritance. We conjecture that in both cases the natural solution is to test for the logical equivalence of the contracts, rather than for simple implications.

4.2 Assigning Blame in General

The strict interpretation of the Liskov-Wing principle of substitutability imposes overly stringent constraints on the architects of class and interface frameworks. According to current OOP practice, software producers deliver such frameworks with pre-defined extension points where customers plug in their own class and interface sub-hierarchies. Programmers who do that can often reason about the entire implementation, though cannot modify the producer-supplied code [3]. Therefore, they can make judgments based on the entire body of code rather than just contracts. In such situations, it is often possible to state contracts that seem to contradict the Liskov-Wing principle but that don't violate it at run-time.

Let us consider a small example. Consider the following two interfaces:

```
interface Sq_root {
  float getValue();

  Number sq_root()
  // @post @ret instanceof Float && @ret.getValue() > 0
}

interface Complex_sq_root extends Sq_root {
  float getValue();

  Number sq_root()
  // @post (@ret instanceof Float && @ret.getValue() > 0)
  //       || (@ret instanceof Complex)
}
```

Each is a wrapper for a float that has a square root operation. Both return the positive square root of positive floats. Objects matching *Sq_root* return *NaN* for negative floats and objects matching *Complex_sq_root* return complex numbers for negative floats.

These interfaces violate the Liskov-Wing principle because the post-condition of *Complex_sq_root* is not a strengthening of the post-condition of *sq_root*, in all possible contexts. Despite that, a programmer may still want to compose a component that produces instances of *Complex_sq_root* with a component that accepts instances of *Sq_root*. In general, since the programmer uses more information than the weak pre- and post-conditions express, this composition may be legal in all contexts that can occur at runtime. In this simple-minded example, the programmer knows that the square root function only produces complex output when the input is negative. Thus, if the programmer can ensure that numbers in the computation stay positive, it is meaningful to compose the *Complex_sq_root* producer with the *Sq_root* consumer.

In general, to compose two components whose contracts are mismatched, the component composer must assume responsibility for the mis-match between the components. In this example, if an instance of *Complex_sq_root* flows into a function that accepts *Sq_root* and the *sq_root* method returns a complex number, the composer must be blamed for the contractual violation.

As discussed in section 2, components in Java are represented as classes and contracts are specified in interfaces. An upcast from one interface to another may change the contracts that an object must meet arbitrarily. This makes assigning blame for contractual violations difficult.

In order to program with the liberal Liskov-Wing principle, we need a semantic framework that assigns blame for contractual violations based on the history of the object's creation and the casts applied to the object. This semantics must track the casts an object has passed through to arrive at each method call in order to determine the contracts that the object must live up to. It must then assign blame for any contractual violation either to the method caller, the method itself, the cast that changed the visible contracts on the object, or the method that created the object with an initial interface.

5 Related Work

There are two approaches to adding contracts to programs. Contracts are either validated at run-time, or used in conjunction with an analysis to prove programs correct. This work is concerned with the former. We explore a mechanism

for adding dynamically checked contracts to object-oriented programming languages. The most relevant work concerns Eiffel [17, 23], Handshake [5], iContract [10], jContractor [8], and JMSAssert [15] and is discussed in detail in earlier sections. Wasserman and Blum [24] also check contracts at run-time, but they are concerned with improving the efficiency of contracts that enforce correctness. They study contracts that probabilistically guarantee correctness.

Others have attempted to use contracts to prove programs correct. Janowski and Mostowski [7] add regular-expression contract specifications to components to prove their correctness. Obayashi et al [18] use ADL to combine specifications with testing to guarantee correctness. Shankar [22] and Owre et al [19] use PVS which combines model checking with contract specifications to prove correctness. Detlefs et al [4] designed extended static checking, a proof system for finding contractual violations in programs.

These techniques are not yet practical. Also, they require more training in logic than most programmers have. First, the programmers must be able to understand what it means when the proof techniques fail and how to fix the proof or the program. Second, the programmers must be able to understand exactly what their theorems state so they know what the proof guarantees.

Rosenblum's work [21] on programming with assertions is also closely related research. Based on his experience of building an assertion preprocessor for C and using it on several projects, he identifies the use of assertions as contracts for functions as critical. Furthermore, he identifies several important classes of contracts, especially, consistency checks across arguments, value dependencies, and subrange (or subset) membership of data.

Parnas [20] was the first to recognize that languages for building large systems must support a linguistic mechanism for components and that components should come with contracts in the form of total correctness assertions. Anna [14] can be understood as a representative implementation of Parnas's proposal on modules.

6 Conclusion

We have presented the ideas of interface contracts and contract violations in the world of Java. Existing contract monitoring tools fail to catch contract violations or assign blame to the wrong program component. We believe that these failures are due to a lack of foundational work on the semantics of contracts and contract violations in an object-oriented world. In response, we have formulated a challenge to the foundational software engineering community and have presented two design spaces for a solution.

References

1. Addy, E. A. A framework for performing verification and validation in reuse-based software engineering. *Annals of Software Engineering*, 5:279–292, 1998. Systematic Software Reuse.
2. Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, june 1999.
3. Bohrer, K. Architecture of the san francisco frameworks. In *IBM Systems Journal*, number 2, pages 156–169, 1998.
4. Detlefs, D. L., K. Rustan, M. Leino, G. Nelson and J. B. Saxe. Extended static checking. Technical Report 158, Compaq SRC Research Report, 1998.
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. Huizing, K., R. Kuiper and SOOP. Verification of object oriented programs using class invariants. In *Third International Conference, FASE 2000*, pages 208–221. Springer-Verlag, LNCS 1783, March 2000.
7. Janowski, T. and W. Mostowski. Fail-stop software components by pattern matching. UNU/IIST Report #164, The United Nations University, May 1999.
8. 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 *Lecture Notes in Computer Science*, July 1999.
9. Knight, J. C. and M. F. Dunn. Software quality through domain-driven certification. *Annals of Software Engineering*, 5:293–315, 1998. Systematic Software Reuse.
10. Kramer, R. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, 1998.
11. Krishnamurthi, S. PLT McMicMac: Elaborator manual. Technical Report 99-334, Rice University, Houston, TX, USA, 1999.
12. Krishnamurthi, S., M. Felleisen and B. F. Duba. From macros to reusable generative programming. In *International Symposium on Generative and Component-Based Software Engineering*, number 1799 in Lecture Notes in Computer Science, pages 105–120, September 1999.

13. Luckham, D. *Programming with Specifications*. Springer-Verlag, 1990.
14. 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.
15. Man Machine Systems. Design by contract for java using jmsassert. http://www.mmsindia.com/DBCForJava.html, 2000.
16. 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.
17. Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.
18. Obayashi, M., H. Kubota, S. P. McCarron and L. Mallet. The assertion based testing tool for OOP: ADL2. In *International Conference on Software Engineering*, 1998.
19. Owre, S., S. Rajan, J. Rushby, N. Shankar and M. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur, R. and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
20. Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
21. Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.
22. Shankar, N. PVS: Combining specification, proof checking, and model checking. In Srivas, M. and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, November 1996. Springer-Verlag.
23. Switzer, R. *Eiffel: An Introduction*. Prentice Hall, 1993.
24. Wasserman, H. and M. Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, November 1997.
25. Weck, W. Inheritance using contracts and object composition. In *Proceedings of the Workshop on Components-Oriented Programming*, 1997.