Jass – Java with Assertions

Detlef Bartetzko [1]

Atanion GmbH
Bismarckstr. 13, D–26122 Oldenburg, Germany

Clemens Fischer [2]

BTC – Business Technology Consulting AG
Industriestr. 9, D–26121 Oldenburg, Germany

Michael Möller [3] Heike Wehrheim [4]

Universität Oldenburg, Fachbereich Informatik
Postfach 2503, D–26111 Oldenburg, Germany

Abstract

Design by Contract, proposed by Meyer for the programming language Eiffel, is a technique that allows run-time checks of specification violation and their treatment during program execution. Jass, **J**ava with **ass**ertions, is a Design by Contract extension for Java allowing to annotate Java programs with specifications in the form of assertions. The Jass tool is a pre-compiler that translates annotated into pure Java programs in which compliance with the specification is dynamically tested. Besides the standard Design by Contract features known from classical program verification (e.g. pre- and postconditions, invariants), Jass additionally supports *refinement*, i.e. subtyping, *checks* and the novel concept of *trace assertions*. Trace assertions are used to monitor the dynamic behaviour of objects in time.

1 Introduction

Correctness is a major issue in the design of large software systems. Correctness concerns the question whether the design or implementation of a system meets the requirements (or specifications) set out in earlier phases of the development. *Modelchecking* [3] of finite state systems is successfully applied

[1] Email: `Detlef.Bartetzko@atanion.com`
[2] Email: `Clemens.Fischer@ewe.de`
[3] Email: `Michael.Moeller@informatik.uni-oldenburg.de`
[4] Email: `Heike.Wehrheim@informatik.uni-oldenburg.de`

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

in many areas to check that systems satisfy their specifications. However, modelchecking often suffers from the so-called state explosion problem which forbids verification because of complexity. Classical *program verification* techniques [9,1] on the other hand often do not scale-up well to large programs. The development of program verification techniques for object-oriented languages (and its support by theorem provers) is a topic of current research (see for instance [12,13,20]).

Design by Contract, as proposed by Meyer for the object-oriented language Eiffel [19], is a lightweight formal technique that allows for dynamic run-time checks of specification violation. The name refers to a contract which is made between the client and the supplier of a component. The contract governs the relation between client objects requesting services (by calling a feature) and supplier objects providing these services. The specification of the contract is directly written into the program in the form of *assertions*. Ideally, the syntax of assertions is close to the programming language itself and thus easy to use for all programmers. Assertions are checked during *program execution* and thus violation is amenable to exception handling (and correction).

Design by Contract extensions (or simply assertions) have been proposed for a number of languages besides Eiffel, for instance for Ada (ANNA, [17]) and C++ (see `http://www.elj.com/eiffel/feature/dbc/java/ge/`), or are directly incorporated into new languages as for instance in Promela [11], the language of the SPIN modelchecker. For Java, there are already a number of proposals for Design by Contract extensions: Sun is currently developing a simple assertion facility for Java [23], and tools like iContract [14] or specification languages like JML [15] or BSL [4] already allow for a rich set of assertions to be written into Java code. In this paper we present the tool Jass [8] which, besides the standard Design by Contract facilities like pre-, postconditions and class invariants, additionally offers *refinement checks* and *trace assertions*. Trace assertions are used to monitor the dynamic behaviour of an object, the ordering of method invocations and calls in time. Thus they are especially useful in the design of *concurrent*, *reactive* systems. The idea behind this concept is the following: while standard assertions are the Design by Contract counterpart of state-based formal specifications, trace assertions are the counterpart of behaviour-oriented specifications like process algebras or temporal logic. The design of trace assertions in Jass is influenced by the process algebra CSP [10]: in the syntax of trace assertions a number of operators close to CSP operators appear. In fact, our ultimate goal is the *generation* of assertions from formal specifications, in our case, from specifications written in CSP-OZ [8,7], a combination of Object-Z and CSP. Trace assertions are a novel concept for Design by Contract and, to the best of our knowledge, none of the above mentioned tools [5] currently support a similar facility.

Refinement checks are also inspired by an important design technique of

[5] Modelcheckers (like Spin or FDR) can, of course, check compliance with respect to specified allowed traces.

formal methods. Refinement [5] usually relates specifications on different levels of abstraction, saying when a concrete class (for instance with very efficient data types) is a correct implementation of a more abstract class. In Jass, refinement checks can be used to check whether a subclass is a *behavioural subtype* (in the sense of [16]) of its superclass. In a subtype, method preconditions can be weakened and postconditions strengthened. Furthermore, the invariant of the superclass has to be preserved in the subclass. A refinement check for subtypes in Jass is carried out once an interface `jass.runtime.Refinement` is implemented in the subclass. The method `jassGetSuperState` plays the role of the usual abstraction or representation function of refinement, relating concrete with abstract states.

Jass is a pre-compiler which itself is written in Java. It translates annotated into pure Java programs in which compliance with the specification is *dynamically* tested during *run-time*. This is in contrast to *static checkers* like ESC [6] or LCLint, which perform a static check of the assertions against the actual program code employing dataflow analysis techniques or, in case of ESC, theorem provers. It is also unlike *modelchecking* approaches for Java like Bandera [4], a tool for translating Java programs into input for a modelchecker and assertions into temporal logic requirements, or Java PathFinder [24], a modelchecker operating directly on Java programs. Here, we are only concerned with run-time checks of specification violation.

Assertions in Jass are written as comments into the Java code, thus every well-formed Jass program is also a well-formed Java program. Assertions are simply boolean expressions of Java extended with certain keywords and (existential and universal) quantification over finite sets. Jass is available free of charge [6] . The tool was implemented as part of three master's thesis' [18,2,21]; the general ideas have been developed in [8], which also gives a formal semantics to Jass.

2 Basic Jass Features

Design by Contract aims at dynamic run-time checks of *specification violations*. To this end the program code is decorated with assertions which provide the specification of the program. Jass translates these assertions into Java code such that violation of the specification is indicated by a Java exception. This section will explain the basic features of Jass referring to the following `Buffer` example. The type of assertions discussed in this section are provided by most of the "Java with Design by Contract"-tools. The next sections discuss advanced features of Jass.

```
public class Buffer implements Cloneable {
    private int in,out,count;
    private Object[] store;
```

[6] and can be found at `http://semantik.Informatik.Uni-Oldenburg.DE/~jass/`

```
    public Buffer (int capacity) { ... }
    public boolean empty() { ... }
    public boolean full() { ... }
    public int capacity() { ... }
    private boolean inRange(int i) { ... }

    public void add(Object o) { ... }
    public Object remove() { ... }

    public boolean contains(Object o) { ... }
    ...
}
```

The `Buffer`[7] stores objects (method `add`), that may be recalled later (method `remove`), on a first-in-first-out basis (FIFO). The `Buffer` has a limited capacity, so that objects can only be stored when the buffer is not already full. Of course, no objects can be returned when the buffer is empty. Method `inRange` is used to check whether an index accessing `store` is in range.

To specify the behaviour of such a buffer through Design by Contract Jass allows to insert different kinds of assertions into the Java source code:

- method pre- and postconditions,
- class invariants,
- loop invariants and variants, and
- additional checks (similar to predicates in proof sketches).

In a Jass source file, assertions are written into the code as special formated comments, so that a well-formed Jass program will always be a well-formed Java program. This may help to get familiar with Jass easily and to minimize the threshold for using Jass in an industrial context: If the Jass precompiler is not suitable in some stage of a project the code can still be used without change. Assertions are boolean expressions[8] for which Jass will insert Java code into its output. In addition to standard Java boolean expressions Jass allows to use universal and existential quantifications that range over finite sets. This feature helps to shorten the description of conditions and makes it possible to state a condition for a varying but finite number of objects. The following can for instance be used in an invariant for class `Buffer`.

Example 2.1 universal quantification

```
forall i : {0 .. capacity()-1} # !inRange(i) || store[i] != null
```

If a boolean expression does not evaluate to true during runtime an `AssertionException` that indicates the violation of the contract is thrown.

[7] The interface `Cloneable` is introduced for technical reasons.
[8] except for the loop variant, that is an integer expression.

To help the developer in debugging the program most assertions may be decorated by labels, that are reused in the exceptions. This facilitates identification of the point of failure.

In the following we will discuss the basic assertions that Jass provides in more detail.

2.1 Pre– and postconditions of methods

A precondition can be used to specify the valid states of method invocation. All states in which the assertion expression is evaluated to true are legal for calling the method. Satisfying the precondition is the duty of the caller. The precondition is checked at the beginning of the methods body. So this is also the place where the precondition must be declared with the introducing keyword `require`.

Example 2.2 precondition of a method

```
public Object remove() {
  /** require !empty(); **/
  ...
}
```

In the precondition the called methods and used variables must be as visible as the method they appear in. This *availability* rule introduced by Bertrand Meyer ensures that the caller can understand the conditions under which the method can be invoked. This implies that e.g. a protected method may only use public or protected members in the precondition.

A postcondition specifies the legal states after method invocation. When leaving the method the postcondition must evaluate to true. Satisfaction of the postcondition is the duty of the developer of the method. The postcondition is checked at all *normal* return points of the methods, that is, all return statements and the end of the method body. The postcondition must be declared at the end of the method body with the introducing keyword `ensure`.

As a special feature of Jass the return value of a method can be accessed in the postcondition with the special variable `Result`. Additionally the objects state at the beginning of the method is stored in the special variable `Old` [9]. Using this variable the developer can specify relations between entry and exit states, for example the monotonicity of a counter.

Another special construct is the `changeonly` keyword followed by a list of attributes. If such a list is specified in a postcondition only the declared attributes are allowed to change their values (a frame condition). This feature is inspired by the Δ-lists of Object-Z.

---

[9] In order to be able to access this facility, the programmer has to implement the interface `java.lang.Cloneable`. The method `clone` of this interface fixes the way copies of states are made.

Example 2.3 postcondition of a method using `Old` and `changeonly`

```
public Object remove() {
    ...
    return o;
    /** ensure changeonly{count,out};
          Old.contains(Result);
          Result.equals(Old.store[Old.out]); **/
}
```

To give just an impression of how the translation into Java is done, the following gives the Java code for checking pre- and postconditions.

```
public Object remove() {

   examples.Buffer jassOld = (examples.Buffer)this.clone();
   /* precondition */
      if (!(!jassInternal_empty()))
          throw new jass.runtime.PreconditionException(...);

   ...
   jassResult = ( o);

  /* postcondition */
      if (!(in == jassOld.in &&
            jass.runtime.Tool.arrayEquals(store,jassOld.store)))
          throw new jass.runtime.PostconditionException(...);
      if (!(jassOld.jassInternal_contains(jassResult)))
          throw new jass.runtime.PostconditionException(...);
      if (!(jassResult.equals(jassOld.store[jassOld.out])))
          throw new jass.runtime.PostconditionException(...);
   return jassResult;
}
```

First, a copy of the object is made (the old value), then the precondition is checked (a precondition exception is thrown if the negation of the precondition holds), the body of the method is executed, where `return` statements are replaced by an assignment to `jassResult`. Finally the postcondition is checked and the result is returned. The postcondition check requires a comparison with the old value of the object which has been stored in `jassOld` and the result of the method call, stored in `jassResult`. The pre- and postcondition checks call special methods `jassInternal_empty` and `jassInternal_contains` which have the same body as methods `empty` and `contains`, however, without the pre- and postcondition checks. The method `jass.runtime.Tool.arrayEquals` tests the equality of the contents of two arrays.

2.2 Class invariant

A class invariant specifies the allowed global states of a class. It expresses restrictions on and relationships between values of the attributes. The class invariant is declared with the introducing keyword `invariant` and is located at the end of the class body. The class invariant is checked whenever the state of the class is stable, i.e. whenever a method of the class is called or ends [10]. Analogous to the postcondition the end of a method are the return statements and the end of the body. No local variables and formal parameters are allowed in the class invariant since assertions have to be evaluable in every method.

Example 2.4 class invariant

```
public class Buffer {
  ...
  /** invariant 0 <= count && count <= capacity(); **/
}
```

2.3 Loop variant and invariant

The purpose of loop variants and invariants is to catch the typical programming errors in loops:

- The loop does not terminate,
- the body of the loop is executed once too often or too less,
- special cases like zero iterations are not handled.

To facilitate programming of loops the loop invariant and the loop variant are introduced. Both are declared after the head and before the body of a loop. The loop invariant starts with the keyword `invariant` and specifies a condition that must be satisfied at the beginning of the loop, after every iteration and when the loop has terminated. This is used to express global properties of the loop like the range of the loop variable. The loop variant has the introducing keyword `variant` and declares an integer expression that is decreased within every loop iteration but limited to not being less than zero. Thus the loop variant guarantees the termination of the loop.

Example 2.5 loops

```
public boolean contains(Object o) {
      for (int i = 0; i < capacity(); i++)
      /** invariant 0 <= i && i <= capacity(); **/
      /** variant capacity() - i **/
         if (inRange(i) && store[i].equals(o)) return true;
      return false;
}
```

[10] If there is no violation of the contract and no external access to attributes, it would be adequate to check only on method termination.

Note that the loop invariant also has to hold after termination of the loop, it thus may only require $i \leq capacity()$ not $i < capacity()$.

2.4 Miscellaneous

There are two more basic facilities in Jass that we will only shortly discuss (see the Jass homepage for a more thorough introduction). These features are *check* statements and *rescue* and *retry* statements. Check statements can appear at all places where a normal Java statement can stand. They are boolean expressions which are evaluated when program control reaches them. Check statements closely match the `assert` statement that Sun is currently adding to Java.

Rescue and retry statements are used to handle exceptions thrown at contract violation. They can be placed at the end of a method body and specify which assertion exceptions should be caught and what code blocks are to be executed then. The indicating keyword is `rescue`. A Jass rescue statement is translated into a Java try-catch-block. Additionally, a special keyword `retry` can be used in the rescue blocks to re-initiate the method call, possibly with changed parameter values.
Next, we describe two more advanced features of Jass.

3 Refinement checks

Refinement checks are used to validate whether a concrete class C is a refinement or *behavioural subtype* of another more abstract class A[11]. Refinement is an important concept for a stepwise design of systems supporting the specification of classes on different abstraction levels. In Jass, refinement checks are carried out in the following way: The programmer has to implement the interface `jass.runtime.Refinement` in the class C. When Jass precompiles such a class, this requires that the method `jassGetSuperState` is implemented in class C returning an object reference of type A. It plays the role of the abstraction or representation relation in refinement relating concrete with abstract state. (Since `jassGetSuperState` always implements a *function* we cannot be as general as representation relations here; thus we only test for *functional refinement*.) Once this interface is implemented in a class, Jass carries out the following *dynamic refinement checks*:

- whenever a state is reached in an object of class C and C's invariant is satisfied then in the corresponding abstract state A's invariant should be satisfied as well, and

[11] This has nothing to do with the Java class modifier `abstract`, which is used for classes that only declare some methods but do not implement them. Here the word "abstract" refers to the more general class in the inheritance hierarchy. These abstract, general classes may however not be abstract in the Java sense since during the refinement check Jass has to be able to create an instance of this class.

- whenever a method m is called in a particular state, the holding of the precondition of m in the corresponding abstract state should imply the holding of the precondition in the concrete state (weakening of preconditions), and
- when returning from a method such that the abstract precondition is fulfilled in the abstract state corresponding to the concrete state at the beginning of the method, then the holding of the concrete postcondition should imply the holding of the abstract postcondition (again only considering this particular state and its abstract counterpart) (strengthening of postconditions).

These are the standard forward simulation rules of refinement (invariant preservation, and applicability and correctness of operations) as for instance known from Z [22] (with the exception that we only dynamically check these conditions and only for those states that are reached during program execution).

We again use the example of the buffer to demonstrate refinement checks. This time consider the `Buffer` to be derived from an abstract version that uses a multi-set with limited capacity to store elements. In the `AbstractBuffer` the remove operation will return an arbitrary element of the multi-set. Since the concrete `Buffer` uses a FIFO strategy for storing, the postcondition of remove is stronger in the concrete class: it specifies that a fixed element is to be returned.

Thus we now consider class `Buffer` to be a subclass of `AbstractBuffer` and indicate that refinement checks should be carried out by implementing the Jass refinement interface. The `jassGetSuperState` implementation will build an `AbstractBuffer` that represents the current state of the concrete buffer by creating a new instance of the superclass with the same capacity and inserting all objects, that are currently stored. The postcondition of remove in `Buffer` is stronger than that in `AbstractBuffer` (and the preconditions are equivalent), thus the refinement check will always succeed.

Example 3.1 refinement

```
public class AbstractBuffer implements Cloneable {

    private LimitedMultiSet buffer;
    ...
    public Object remove() {
        /** require !empty(); **/
                ...
        /** ensure Old.contains(Result); **/
    }
    ...
}


public class Buffer extends AbstractBuffer
```

```
                    implements jass.runtime.Refinement {
  ...
  public Object remove() {
    /** require !empty(); **/
            ...
    /** ensure changeonly{count,out};
          Old.contains(Result);
          Result.equals(Old.store[Old.out]); **/
  }
  ...
  private AbstractBuffer jassGetSuperState() {
    int capacity = store.length;
    AbstractBuffer aState = new AbstractBuffer(capacity);
    for (int i = capacity + in - count; i < capacity + in; i++)
        aState.add(store[i % capacity]);
    return aState;
  }
}
```

We shortly discuss two other approaches to subtyping. Eiffel adopts the following principle for subcontracting: in the subclass pre- and postconditions of redeclared methods are combined with the corresponding pre- and postconditions of the superclass. Preconditions are combined with a logical *or* and postconditions with *and* thus achieving a weakening of pre- and a strengthening of postconditions. State spaces of sub- and superclass have to be identical (at least, concerning the attributes accessed in the assertions); representation functions cannot be applied. JML [15] uses a more elaborate treatment of subtyping which is close to what Jass supplies. A `represents` clause can be used to relate concrete with abstract state space, and class invariants and method specifications are inherited from the superclass. Furthermore, the concept of history constraints from [16] is included. In JML, subtyping relationships are however mainly used to facilitate the specification of the subtype in that assertions can be inherited from the superclass.

Another difference between Jass and JML can be found at the following point. JML uses a concept of model variables in specifications: model variables are no actual variables of classes, but are only used for specification purposes. Pre- and postconditions can then be written refering to model variables. Since this allows for a clearer separation of specification and code we currently think of adopting a similar approach in Jass.

4 Trace Assertions

To specify the intended dynamical behaviour Jass uses a CSP like notation for describing allowed traces of events. In the context of Jass events are beginnings and ends of method invocations. Consequently events are written like method invocations followed by `.b` or `.e` to indicate the method entry or exit event. A method invocation without those suffixes abbreviates the entry event followed by the exit event. The allowed traces are specified by CSP-like processes. Jass processes are defined by a process name, parameters of the process (like formal parameters of Java methods), local variables and a process expression. The following example gives an impression of trace assertions in Jass. The class `Factorial` will calculate the factorial of a positive integer value by recursively calling the method `factorial`. The process of the trace assertion is used to monitor the correct invocation of this method: Each recursive invocation of `factorial` will initiate a process, that will only accept another call to `factorial` when the parameter has been reduced but is still not less than zero. Since trace assertions describe the observable behaviour of a class they have to be declared as class invariant. The generated code will check, that the trace of the current program execution is included in the traces of the `MAIN` process, otherwise a trace exception is thrown.

Example 4.1 observing recursive method calls

```
public class Factorial {
  public int factorial(int value) {
    /** require value > 0; **/
    if(value == 1)
      return 1;
    else
      return value * factorial(value - 1);
    /** ensure Result > 0; **/
  }

  /** invariant [variant] trace (
        MAIN() {
          int value;
          factorial(?value).b -> CALL Decrease(value)
        }

        Decrease(int variant) {
          int nextVariant;
          IF(variant < 0) {
              EXECUTE(throw new
                RuntimeException ("negative method variant!");)
              -> STOP
          } ELSE {
              -> factorial(?nextVariant).b
                                 WHERE(nextVariant < variant)
              -> CALL Decrease(nextVariant)
```

```
            }
          }
        );
  **/
}
```

A number of features in trace assertions can be seen in this example. The trace assertions defines two processes (`MAIN` and `Decrease`), the second one is parameterised. Both processes have local variables (`value` and `nextVariant`), which are used as input variables in events, i.e. the event `factorial(?value).b` matches a method call of `factorial` with any value of parameter `value`, and with the execution the current value of the parameter is bound to the local variable `value`. Analogously output variables can be used (prefixing a local variable with !). The allowed values for input variables can be restricted by a `WHERE` clause. Furthermore, the example already shows the use of a number of CSP operators: prefixing `->`, `IF-ELSE`, process calls (with `CALL`) and the process `STOP`. Jass additionally allows processes `ANY` and `TERM` accepting any event or just the successful termination of a class, respectively, and operators for *external choice* and *parallel composition.* Parallel composition is restricted to synchronisation on the intersection of the components alphabets, thus ensuring the specification of *deterministic* processes only. The `EXECUTE` event in the example is a special feature of Jass that allows the developer to execute Java statements during trace testing.

During the check of trace assertions only those events are monitored which are in the *alphabet* of the trace assertion. This alphabet is implicitly given by the trace assertion (the set of events occurring in it) or can be explicitly stated.

5 Additional Features (in prototype state)

Jass provides two additional features, that are, however, so far only rudimentary implemented. The features do work in the described way but there are still some limitations that should be removed.

Interference Checks Jass provides a simple facility to detect possible interferences in a parallel program. The threads, that run in parallel, must be Jass classes and must be started by a `main` method. In this case Jass is able to detect when assertions in one thread may become invalid through statements in another thread. Currently Jass is not able to detect whether those possible interferences are guarded by synchronized statements, neither whether the statement and assertion refers to the same object instance.

JavaDoc Support Jass is able to add the pre- and postcondition or class invariant to a JavaDoc conform comment of a method or class. Currently this is done by adding HTML code for these assertions to the JavaDoc comment in the generated Java source code.

6 Conclusion

In this paper we presented the tool Jass, a precompiler for annotated Java programs. The assertion language of Jass allows all standard Design by Contract assertions to be written into programs, and additionally supports the novel concept of trace assertions and refinement checks.

As future work we envisage the *automatic generation* of assertions from formal specifications, in our case from specifications written in CSP-OZ [7], a formal method combining CSP with Object-Z.

Concerning the Jass precompiler, we plan to change the translation in such a way that Jass assertions are directly translated into the new `assert` statement of Java. There, evaluation of assertions is fixed at the time the class is loaded, and thus may be enabled for certain classes and disabled for others.

Acknowledgements. Many thanks to Michael Plath who implemented trace assertions within Jass, Dieter Meemken for the first version of an assertion precompiler and Jochen Hoenicke for general support concerning Java.

References

[1] Apt, K.-R. and E.-R. Olderog, "Verification of Sequential and Concurrent Programs," Springer-Verlag, 1997, 2nd edition.

[2] Bartetzko, D., "Parallelität und Vererbung beim "Programmieren mit Vertrag"," Master's thesis, Universität Oldenburg (1999), in german.

[3] Clarke, E., E. Emerson and A. Sistla, *Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach*, in: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1983, pp. 117–126.

[4] Corbett, J., M. Dwyer, J. Hatcliff and Robby, *A language framework for expressing checkable properties of dynamic software*, in: *SPIN 2000*, number 1885 in LNCS (2000), pp. 205 – 223.

[5] de Roever, W.-P. and K. Engelhardt, "Data Refinement – Model-Oriented Proof Methods and their Comparison," Cambridge University Press, 1998.

[6] Detlefs, D., R. Leino, G. Nelson and J. Saxe, *Extended static checking*, Technical Report 159, Compaq Systems Research Center (1998).

[7] Fischer, C., *CSP-OZ: A combination of Object-Z and CSP*, in: H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, IPIF **2** (1997), pp. 423–438.

[8] Fischer, C., "Combination and Implementation of Processes and Data: From CSP-OZ to Java," Ph.D. thesis, University of Oldenburg (2000).

[9] Hoare, C., *An axiomatic basis for computer programming*, Communications of the ACM **12** (1969), pp. 576–580.

[10] Hoare, C. A. R., "Communicating Sequential Processes," Prentice Hall, 1985.

[11] Holzmann, G. J., "Design and Validation of Computer Protocols," Prentice Hall, 1990.

[12] Huisman, M. and B. Jacobs, *Java program verification via a hoare logic with abrupt termination*, in: T. Maibaum, editor, *FASE 2000: Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science **1783** (2000), pp. 284 – 303.

[13] Huzing, K., R. Kuuiper and SOOP, *Verification of object-oriented programs using class invariants*, in: T. Maibaum, editor, *FASE 2000: Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science **1783** (2000), pp. 208 – 221.

[14] Kramer, R., *iContract - the Java Design by Contract tool*, Technical report, Reliable Systems (1998), `http://www.reliable-systems.com`.

[15] Leavens, G., A. Baker and C. Ruby, *Preliminary design of JML: A behavioral interface specification language for java*, Technical report, Department of Computer Science, Iowa State University (1998, revised 2001).

[16] Liskov, B. and J. Wing, *A behavioural notion of subtyping*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 1811 – 1841.

[17] Luckham, D., F. von Henke, B. Krieg-Brückner and O. Owe, "ANNA - A language for annotating Ada programs," Lecture Notes in Computer Science **260**, Springer, 1987.

[18] Meemken, D., "Programmieren mit Vertrag in Java," Master's thesis, Universität Oldenburg (1997), in german.

[19] Meyer, B., "Object-Oriented Software Construction," ISE, 1997, 2nd edition.

[20] Müller, P. and A. Poetzsch-Heffter, *Modular specification and verification techniques for object-oriented software components*, in: G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, Cambridge University Press, 2000 (to appear).

[21] Plath, M., "Trace Zusicherungen in Jass - Erweiterung des Konzepts "Programmieren mit Vertrag" ," Master's thesis, Universität Oldenburg (2000), in german.

[22] Spivey, M., "The Z Notation — A Reference Manual." Prentice Hall, 1992, 2nd edition.

[23] Sun Microsystems, *A simple assertions facility for the Java programming language*, `http://jcp.org/jsr/detail/41.jsp`.

[24] Visser, W., G. Brat, K. Havelund and S. Park, *Model checking programs*, in: *International Conference on Automated Software Engineering*, 2000.