Checking the Conformance of Java Classes Against Algebraic Specifications

Isabel Nunes, Antónia Lopes, Vasco Vasconcelos, João Abreu, and Luís S. Reis

Faculty of Sciences, University of Lisbon, Campo Grande, 1749–016 Lisboa, Portugal
{in,mal,vv,joao.abreu,lmsar}@di.fc.ul.pt

Abstract. We present and evaluate an approach for the run-time conformance checking of Java classes against property-driven algebraic specifications. Our proposal consists in determining, at run-time, whether the classes subject to analysis behave as required by the specification. The key idea is to reduce the conformance checking problem to the runtime monitoring of contract-annotated classes, a process supported today by several runtime assertion-checking tools. Our approach comprises a rather conventional specification language, a simple language to map specifications into Java types, and a method to automatically generate monitorable classes from specifications, allowing for a simple, but effective, runtime monitoring of both the specified classes and their clients.

1 Introduction

The importance of formal specification in software development is widely recognized. Formal specifications are useful for developers to reuse existing software. They also help programmers in understanding what they have to provide. Furthermore, they can be used as test oracles, i.e., system behavior can be checked against the specification.

Currently, Design by Contract (DBC) [18] is the most popular approach for formally specifying OO software. In this approach, specifications are class interfaces (Java interfaces, Eiffel abstract classes, etc.) annotated with pre/post conditions pairs expressed in a particular assertion language. At runtime, the implementation can be tested against its specification by means of contract monitorization.

Although the DBC methodology has become very popular, programmers rarely specify contracts—the strong restrictions to the kind of properties that are both expressible and monitorable, contribute to the frustration of being left with very poor specifications. Furthermore, as argued by Barnett and Schulte [3], contract specifications do not allow the level of abstraction to vary and do not support specifying components independently of the implementation language and its data structures.

Algebraic specification [2, 6, 10] is another well-known approach to the specification of software systems that supports a higher-level of abstraction. Algebraic approaches can be divided into two classes: *model-oriented* and *property-driven*.

From the two, *model-oriented* approaches to specification, like the ones promoted by Z [20], Larch [11] and JML [17], definitely prevail within the OO community. In most of these approaches, the behavior of a class is specified through a very abstract implementation, based on primitive elements available in the specification language.

Implementations can be tested against specifications by means of runtime assertion-checking tools. This requires an *abstraction function* to be explicitly provided. In JML, for instance, a concrete implementation is expected to include JML code defining the relation between concrete and abstract states. Although we recognize the important role played by model-based approaches, we believe that, for a significant part of programmers, understanding or writing this kind of specifications can be rather difficult. Moreover, programmers implementing a specification have to define the appropriate abstraction mapping, which can also be rather difficult to obtain.

In contrast, for a certain class of programs, in particular for *Abstract Data Types* (ADTs), *property-driven* specifications [6, 8] can be very simple and concise: the observable behavior of a program is specified simply in terms of a set of abstract properties. The simplicity and expressive power of property-driven specifications may encourage more programmers to use formal specifications. However, the support for checking OO implementations against property-driven specifications is far from being satisfactory. As far as we know, it is restricted to previously-presented approaches [1, 13], whose limitations are discussed in detail in Section 8.

This paper presents a new approach for runtime checking OO implementations against property-driven specifications. The key idea is to reduce the problem to the runtime monitoring of contracts, which is supported by many runtime assertion checking tools (e.g., [5, 15–17, 21]). The classes under testing become wrapped by automatically generated classes. The wrapper classes are annotated with run-time checkable contracts automatically generated from the corresponding specifications.

A distinguishing feature of the approach is that our module specifications not only specify behavioral properties required from implementations, but they also define the required architecture of the implementations, i.e., how the implementation should be structured in terms of classes. This is important to support reuse: it allows to enforce that the implementation of a module M is achieved in terms of classes that can be reused in the implementation of other modules that have elements in common with M.

The approach is tailored to Java and JML [17] but it could as well be defined towards other OO programming and assertion languages (or other programming languages with integrated assertions [4, 18]). It comprises a specification language that allows automatic generation of JML contracts, and a language for defining refinement mappings between specification modules and collections of Java classes. Refinement mappings define how sort names are mapped to class names and operation signatures are mapped to method signatures. Because this activity does not require any knowledge about the concrete representation of data types or component states, refinement mappings are quite simple to define. Our approach offers several benefits. More significantly:

- Specifications are easier to write and understand since they are written in a more abstract, implementation independent, language. The same applies to refinement mappings, whose definition does not require any knowledge about the concrete representation of data types or component states, as happens for instance in JML.
- Several Java classes or packages can be tested against the same specification. This contrasts with, for example, the JML approach in which different implementations may require different JML specifications. For instance, JML contract specifications appropriate for immutable classes are not suitable for mutable classes.

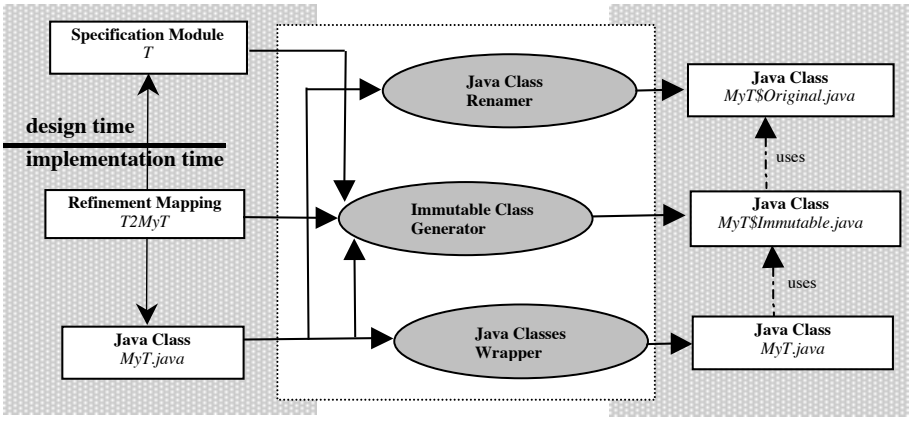

Fig. 1. Approach overview

– The same Java class or package can be tested against several specifications without requiring any additional effort.

In Section 2 we present a quick overview of our approach. Section 3 describes the structure of specification modules and our specification language. In Section 4 the world of specifications and that of implementations are related through the notion of refinement mappings. In Section 5 we describe the wrapper and immutable classes equipped with contracts that are generated, and illustrate their use through an example. In Section 6 we focus on the methodology for generating contracts from specification axioms. Section 7 reports on the results of our experiments. Section 8 presents related work, and Section 9 concludes, describing limitations of our approach, and topics that need further work.

2 Approach Overview

The process of checking the conformance of a collection of Java classes against a specification module consists in inspecting, during execution, the variances between actual and required behavior. This presumes that the implementation is structurally consistent with the specification module which, in our approach, means that there is a *refinement mapping* between the module specification and the collection of Java classes, defining which class implements each sort, and which method implements each operation.

For the purpose of this overview, we start by considering a simplified scenario in which we want to check a single Java class MyT against a single specification *T*. In this case, the user must supply the refinement mapping defining the relationship between the operation and predicate symbols of *T* and the method names of MyT.

Figure 1 illustrates the several entities involved in our approach. The left part includes the entities that the user must supply. The right part shows the classes that are generated—MyT$Original which is just MyT after renaming, MyT$Immutable equipped with the contracts generated from the axioms in specification *T*, and finally MyT which

is the generated wrapper class. This last one uses MyT$Original, and MyT$Immutable in order to achieve validation, by contract inspection, of the results of invoking the original methods.

The approach consists in replacing class MyT by an automatically generated *wrapper class*. The wrapper class is client to other classes, automatically generated from specification *T*, one of which annotated with contracts. In this way, during the execution of a system involving classes that are clients of MyT, we have that:

1. The behavior of MyT objects is checked (monitored) against specification T;
2. The correctness of clients' behavior with respect to MyT operations is monitored;

provided that the system is executed under the observation of a contract monitoring tool. In both cases violations are reported. Underlying these conditions are the following notions of correction, applicable whenever consistency between class MyT and specification *T* is ensured by the existence of a refinement mapping.

Behavioral correctness. This condition assumes the following notion of behavioral correctness of class MyT with respect to the specification: class MyT is correct if every axiom of *T* (after the translation induced by the refinement mapping) is a property that holds in every execution of a system in which MyT is used. Consider, for instance, that *Stack* is a specification of stacks including the axiom $pop(push(s, e)) = e$ and that MyStack is an implementation of integer stacks with methods **void** push(**int**) and **int** pop(). Class MyStack is a correct implementation of specification *Stack* only if the property **let** t = s **in** (s.push(i); s.pop(); s.equals(t)) holds for all objects MyStack during their entire life. Axiom translation is addressed in detail in Section 6.

Client's correction. This condition relies on a notion of correctness targeted at the clients' classes. As we shall see in Section 3, specifications may include conditions under which the interpretations of some operations are required to be defined. These are called the *domain* conditions. A client class is a correct user of MyT, if it does not invoke MyT methods in states that do not satisfy the domain conditions of the corresponding operations. For instance, if *Stack* is a specification of stacks including a domain condition saying that operation $pop(s)$ is required to be defined if $not\ isEmpty(s)$, then a class C is a correct client only if it never invokes method pop on objects o of type MyStack, such that o.isEmpty() is true.

As mentioned before, a class is generated that has the same name as the original one—MyT. This class has exactly the same interface as, and their objects behave the same as those of, the original MyT class, as far as any client using MyT objects can tell. The generated MyT class is what is usually called a *wrapper class* because each of its instances hides an instance of the original MyT class, and uses it when calling the methods of an immutable version of MyT—the generated class MyT$Immutable—in response to client calls. MyT clients must become clients of the wrapper instead. To avoid modifying them, the original class MyT is renamed—its name is postfixed with $Original— making the wrapping of the original class transparent to client classes.

MyT$Immutable is the class that gets annotated with contracts automatically generated from specification *T*: pre-conditions are generated from domain conditions, and

```
import IntegerSpec
sort IntStack
operations and predicates
  constructors
    clear: IntStack ⟶ IntStack;
    push: IntStack Integer ⟶
                              IntStack;
  observers
    top: IntStack ⟶? Integer;
    pop: IntStack ⟶? IntStack;
    size: IntStack ⟶ Integer;
  derived
    isEmpty: IntStack;
domains
  s: Stack;
  top(s), pop(s) if not isEmpty(s);
axioms
  s: Stack; i: Integer;
  top(push(_, i) = i;
  pop(push(s, _)) = s;
  size(clear(_)) = zero(_);
  size(push(s, _)) = suc(size(s));
  isEmpty(s) iff size(s) = zero(_);
```

```
sort Integer
operations and predicates
  constructors
    zero: Integer ⟶ Integer;
    suc: Integer ⟶ Integer;
    pred: Integer ⟶ Integer;
  observers
    lt: Integer ⟶ Integer;
axioms
  i, j: Integer
  lt(zero(_), suc(zero(_)));
  lt(suc(i), suc(j)) if lt(i, j);
  lt(pred(zero(_)), zero(_));
  lt(pred(i), j) if lt(i, j);
  lt(pred(suc(i)), i);
  pred(suc(i)) = i;
  suc(pred(i)) = i;
```

Fig. 2. Specification of (a) integer stacks, (b) integers.

post-conditions from the axioms that give semantics to the specification operations. Monitoring these contracts correspond to checking (i) whether the properties obtained by translating the specification axioms hold in some particular situations, for some particular objects (these are determined by the contract generation process which is described in Section 6), and (ii) whether client objects do not invoke methods in states that do not satisfy the domain conditions. The fact that MyT$Immutable is, by construction, immutable is essential to ensure that the contracts that are generated are monitorable. Section 5 describes the generated wrapper and immutable classes in more detail.

This approach overcomes the limitations of the direct use of DBC that were mentioned in the introduction. All properties are expressible and monitorable because they are translated into pre- and post-conditions involving only calls to methods that do not change the objects under monitorization.

3 Specifications and Modules

The specification language is, to some extent, similar to many existing languages. In general terms, it supports the description of partial specifications with conditional axioms. It has, however, some specific features, such as the classification of operations in different categories, and strong restrictions on the form of the axioms. It was conceived so that conformance checking with respect to OO implementations can be supported through run-time monitoring of automatically derived contracts. Figure 2 a) presents a typical example in this setting [1, 13, 14]: the ADT integer stack. Figure 2 b) illustrates a specification for integers.

A specification defines exactly *one* sort and the first argument of every operation and predicate in the specification must belong to that sort. Furthermore, operations are classified as ***constructors***, ***observers*** or ***derived***. These categories comprise, respectively, the operations from which all values of the type can be built, the operations that provide fundamental information about the values of the type, and the redundant (but potentially useful) operations. Predicates can only be classified as either ***observers*** or ***derived***.

Specifications are partial because operation symbols declared with $-->?$ can be interpreted by partial functions. In the section ***domains***, we describe the conditions under which interpretations of these operations are required to be defined. For instance, in the specification of integer stacks, both *top* and *pop* are declared as partial operations. They are, however, required to be defined for all non empty stacks.

As usual in property-driven specifications, other properties of operations and predicates can be expressed through axioms, which in our case are closed formulæ of first-order logic restricted to the following specific forms:

- $\forall \vec{y}(\phi \Rightarrow op'_c(op_c(\vec{x}), \vec{t}) = t)$ (relating constructors)
- $\forall \vec{y}(\phi \Rightarrow op_o(op_c(\vec{x}), \vec{t}) = t), \forall \vec{y}(\phi \Rightarrow pred_o(op_c(\vec{x}), \vec{t}))$, $\forall \vec{y}(\phi \Rightarrow \neg pred_o(op_c(x), \vec{t}))$ (defining the result of observers on constructors)
- $\forall \vec{y}(\phi \Rightarrow op_d(\vec{x}) = t), \forall \vec{y}(\phi \Rightarrow pred_d(\vec{x})), \forall \vec{y}(\phi \Rightarrow \neg pred_d(\vec{x}))$ (describing the result of derived operations/predicates on generic instances of the sort).
- $\forall \vec{y}(\phi \Rightarrow x = x')$ (pertaining to sort equality).

where $\vec{y}, \vec{x}$ are lists of variables, x, x' are variables, ϕ is a quantifier free-formula, $\vec{t}$ is a list of terms over $\vec{y}$, t is a term over $\vec{y}$. We use the indexes c, o, d to indicate the kind of operations and predicates that are allowed (constructors, observers, derived).

Notice that, because operations may be interpreted by partial functions, a term may not have a value. The equality symbol used in the axioms represents strong equality, that is to say, either both sides are defined and are equal, or both sides are undefined.

The structure of axioms that is imposed is not only intuitive and easy to understand and to apply, but it is also effective in driving the automatic identification of contracts for classes. In what concerns the expressive power of the language, it is only limited by the fact that we require the sort of the first argument of every operation and predicate in a specification to be the sort introduced in that specification. This rule forces a specific method of organizing specifications which, per se, does not constitute a limitation in the expressive power of the language. The problem is that the rule forbids specifications with constant constructors to be described. Although these constructors are prevalent in algebraic specifications, this limitation has short impact in our approach because it is not possible to provide OO implementations for 0-ary constructors in terms of object methods. Object creation, with a default initialization, is natively supported by OO languages and is not under the control of programmers. These can only define constructors (which are not methods) overriding the default initialization.

Specifications may declare, under ***import***, references to other specifications, and may use external symbols, i.e., sorts, operations and predicates that are not locally declared. For instance, the specification of integer stacks imports *IntegerSpec* and uses sort *Integer* and operation symbols *zero* and *suc*, which are external symbols. Notice

```
public class IntArrayStack implements Cloneable {
  private static final int INITIAL_CAPACITY = 10;
  private int [] elems = new int [INITIAL_CAPACITY];
  private int size = 0;
  public void clear() { size = 0; elems = new int [INITIAL_CAPACITY]; }
  public void push(int i) {
   if (elems.length == size) reallocate();
   elems[size++] = i;
  }
  public void pop() { size--; }
  public int top() { return elems[size - 1]; }
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public boolean equals (Object other) { ... }
  public Object clone() { ... }
  private void reallocate() { ... }
}
```

Fig. 3. Java implementation of an integer stack.

that the specification of integers is self-contained since it does not import any specification. We call it a *closed specification*.

The meaning of external symbols is only fixed when the specification is embedded, as a component, in a *module*. A *module* is simply a surjective function from a set N (of names) to a set of specifications, such that, for every specification: *(i)* the referenced specification names belong to N and *(ii)* the external symbols are provided by the corresponding specifications in the module. The set N defines the set of components of the module. For instance, by naming the two specifications presented in Figure 2a) and b) as *IntStackSpec* and *IntegerSpec*, respectively, we obtain a module *IntegerStack*.

4 Refinement Mappings

In order to check Java classes against specification modules, a user of our approach must supply a *refinement mapping* that bridges the gap between the two worlds. These mappings provide the means for explicitly defining which class implements each type and which method implements each operation and predicate.

A *refinement mapping* $\mathcal{R}$ between a module and a collection of Java classes identifies the type (class or primitive) that implements each module component, as well as the binding between the operations and predicates of the corresponding specification in the module and the methods of the class. Only closed specifications can be implemented by primitive types. Furthermore, bindings are subject to some constraints: predicates must be bound to methods of type boolean; every $n+1$-ary operation or predicate $opp(s, s_1, \ldots, s_n)$ must be bound to an n-ary method $m(t_1, \ldots, t_n)$ such that t_i is the type of the class that, according to $\mathcal{R}$, implements sort s_i. Furthermore, for components that are implemented by primitive types, the binding defines how operations and predicates are expressed in terms of built-in Java operations. Within the structure imposed by the specification, there are several implementation styles that can be adopted.

For instance, the most common implementation of stacks in Java is through a class such as IntArrayStack, presented in Figure 3, where instance methods provide the pred-

```
IntegerSpec is primitive int
  zero(x: Integer): Integer is 0;
  suc(x: Integer): Integer is x + 1;
  pred(x: Integer): Integer is x − 1;
  lt(x: Integer, y: Integer) is x < y;
IntStackSpec is class IntArrayStack
  clear(s: IntStack): IntStack is void clear();
  push(s: IntStack, e: Elem): IntStack is void push(int e);
  pop(s: IntStack): IntStack is void pop();
  top(s: IntegerStack): Elem is int top();
  size(s: IntStack): Integer is int size();
  isEmpty(s: IntStack) is boolean isEmpty();
```

Fig. 4. An example of a refinement mapping.

icates and mutable implementations for operations. In this case, the first argument of operations and predicates is implicitly provided—it is the target object of the method invocation—and the application of an operation whose result type is IntegerStack induces a state change of the current object. Methods implementing these operations are usually procedures (**void** methods) but in some cases programmers decide that the method should also return some useful information about the object (for example, the pop method in the Sun's JDK java.util.Stack class, returns the top element). Although less common, stacks can also be implemented by immutable classes. The difference in this case is that the methods that implement the operations whose result type is IntegerStack return an object of the class; the state change in the current object, if it exists, is not relevant. Another dimension of variability in the implementation of the *IntegerStack* module is related to the choice of the implementation for integers: there is still the possibility of choosing a (Java) primitive type to implement the type.

An admissible refinement mapping for the module *IntegerStack* is presented in Figure 4. It expresses the fact that specification *IntStackSpec* is implemented by the class IntArrayStack whereas sort *Integer* is implemented by the Java primitive type **int**.

Refinement mappings are quite simple to define because they only involve the interfaces of Java classes, that is to say, no knowledge about the concrete representation of data types or component states is needed. The independence from concrete representation makes it possible to test several Java classes or packages against a same specification module—we just have to create the corresponding refinement mappings. Contracts are automatically generated. The approach also allows a refinement mapping to define a mapping from two different components into the same type (class or primitive). This promotes the writing of generic specifications that can be reused in different situations.

5 The Architecture of Wrapped Implementations

As explained in Section 2, our approach for checking Java implementations against specifications comprises wrapping these classes with other, automatically generated, classes. In this section we describe this process in more detail.

Let again *T* be a specification, MyT a given class, and *MyRef* describe a refinement mapping between specification *T* and class MyT. From these, a series of classes are

generated that allow a client class ClientC to invoke methods of MyT while checking whether MyT correctly implements T. Remember that the wrapper class gets its name from the original MyT class, while this is renamed to MyT$Original.

The immutable class equipped with contracts. For each MyT method **void** m($\vec{p}$), class MyT$Immutable defines a **static** method:

```
static MyT$Original m (MyT$Original o, p⃗) {
  MyT$Original aClone = (MyT$Original) clone(o);
  aClone.m(p⃗);
  return aClone;
}
```

and for each MyT method SomeType m($\vec{p}$), class MyT$Immutable defines a method:

```
static SomeType$Pair m (MyT$Original o, p⃗) {
  MyT$Original aClone = (MyT$Original) clone(o);
  return new SomeType$Pair(aClone.m(p⃗), aClone);
}
```

where SomeType$Pair is a generated class that declares two public final attributes—MyT$Original state and SomeType value—and a constructor that receives the values to initialize those attributes. Contracts are generated for the methods in MyT$Immutable that are a translation (see Section 6) of the axioms of the corresponding specification.

The wrapper class MyT defines a single attribute MyT$Original wrappedObject, implements each method **void** m($\vec{p}$) with the following code:

```
{wrappedObject = MyT$Immutable.m(wrappedObject, p⃗);}
```

and implements each method SomeType m($\vec{p}$) with the following code:

```
{SomeType$Pair pair = MyT$Immutable.m(wrappedObject, p⃗);
 wrappedObject = pair.state;
 return pair.value;
}
```

The wrapper class uses the value part of the pair to return the value to the client, and retains the state part in its only attribute (in order to account for methods that, in addition to returning a value, also modify the current object).

Whenever a class, client to the original class, is executed within the context of this framework, every call to a method m in the original class is monitored since the wrapper redirects the call through the corresponding method in the immutable class, forcing the evaluation of the pre and post-conditions. These are such that the original methods behavior is monitored without any side effect on the objects created by client classes.

An Example. Figures 5, 6, and 7 illustrate the classes that are generated according to our approach in the context of the Stack example used throughout the paper—the specification in Figure 2, the class in Figure 3, and the refinement mapping in Figure 4. Consider the following code snippet in a client of IntArrayStack class:

```
IntArrayStack s = new IntArrayStack();
s.push(3);
```

Figures 8 and 9 present UML interaction diagrams showing the interaction between the several objects that participate in the realization of the above instructions. The value of aClone (the return value of the push(stack, 3) operation invoked in 2.1, Figure 9) is monitored by checking the contracts associated with method push in class IntArrayStack$Immutable. The post-condition of push invokes methods size, top, and

```
public class IntArrayStack implements Cloneable {
  private IntArrayStack$Original stack = new IntArrayStack$Original();
  public void clear() { stack = IntArrayStack$Immutable.clear(stack); }
  public void push(int i) { stack = IntArrayStack$Immutable.push(stack, i); }
  public void pop() { stack = IntArrayStack$Immutable.pop(stack); }
  public int size() {
    int$Pair pair = IntArrayStack$Immutable.size(stack);
    stack = pair.state;
    return pair.value;
  }
  public int top() {
    int$Pair pair = IntArrayStack$Immutable.top(stack);
    stack = pair.state;
    return pair.value;
  }
  ...
}
```

Fig. 5. Partial view of the wrapper class that results from applying our approach to the specification IntStackSpec and the original IntArrayStack class.

```
public class int$Pair {
  public final int value;
  public final IntArrayStack$Original state;
  public int$Pair (int value, IntArrayStack$Original state) {
    this.value = value; this.state = state;
  }
}
```

Fig. 6. The auxiliary class int$Pair composed of an integer and an original IntArrayStack.

pop of the immutable class on object aClone. These methods do not change object aClone since they invoke the original versions of size, top, and pop on a clone of aClone. The contracts for these methods are not monitored: the contracts of methods invoked from contracts are not monitored—this is a feature of JML, crucial to our approach since it prevents infinite invocation chains.

This example also illustrates what happens in situations where a primitive type is chosen to implement a specification (in the example, integers are implemented by **int**). While monitoring a given implementation for a module, situations may arise where a class MyT is accused of not correctly implementing specification *T* because a closed specification *T'* was mapped to a Java primitive type that does not correctly implement it. This is due to the fact that our approach does not check the conformance of specifications that are mapped into primitive types. However, this problem can only be overcome with client-invasive approaches, i.e., that require the modification of client classes.

We experienced the situation just described in one of the modules we used to evaluate our approach—a Rational module consisting of specifications of integers and rational numbers, available elsewhere [7]. A traditional immutable implementation of rationals was found to be incorrect during the manipulation of fractions with large numerators and denominators, involving cross-products greater than $2^{31} - 1$. This is due to the fact that Java **int** type does not correctly implements the integer specification,

```
public class IntArrayStack$Immutable {
  //@ ensures size(\result).value == 0;
  static public IntArrayStack$Original clear (IntArrayStack$Original s) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    aClone.clear();
    return aClone;
  }
  //@ ensures size(\result).value == size(s).value + 1;
  //@ ensures top(\result).value == i;
  //@ ensures equal(pop(\result).state, s).value;
  static public IntArrayStack$Original push (IntArrayStack$Original s, int i) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    aClone.push(i);
    return aClone;
  }
  //@ requires ! isEmpty(s).value;
  static public IntArrayStack$Original pop (IntArrayStack$Original s) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    aClone.pop();
    return aClone;
  }
  static public int$Pair size (IntArrayStack$Original s) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    return new int$Pair (aClone.size(), aClone);
  }
 //@ ensures (* See Section 6 *);
    static public boolean equals (IntArrayStack$Original s, Object t) {
    IntArrayStack$Original aClone = (IntArrayStack$Original) clone(s);
    return new boolean$Pair (aClone.equals(t), aClone);
  }
  ...
}
```

Fig. 7. Partial view of the immutable class that results from applying our approach to the *IntegerStack* module and the original IntArrayStack class.

namely properties such as $(i < suc(i))$ and $(n \neq 0 \ \wedge \ m \neq 0 \Rightarrow \ n \times m \neq \ 0)$ do not hold (for example, $2^{31} - 1 + 1$ is a negative number).

6 Contract Generation

In this section we discuss how contracts are generated from specifications. This process can be described in two parts: translation of domain-specific properties described by axioms, and translation of generic properties of equational logic.

6.1 From Axioms to Contracts

Contract generation that captures the properties that are explicitly specified in a given specification *T*, is such that:

- a domain restriction for an operation *op* generates a pre-condition for the method that implements *op*;
- axioms which relate constructors op_c and op'_c, and axioms that specify the result of observers on a given constructor op_c generate post-conditions for the method that implements op_c;

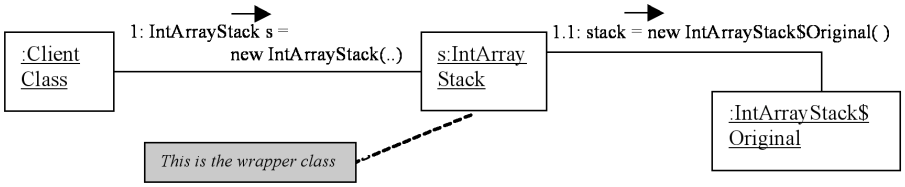

Fig. 8. Creating an IntArrayStack.

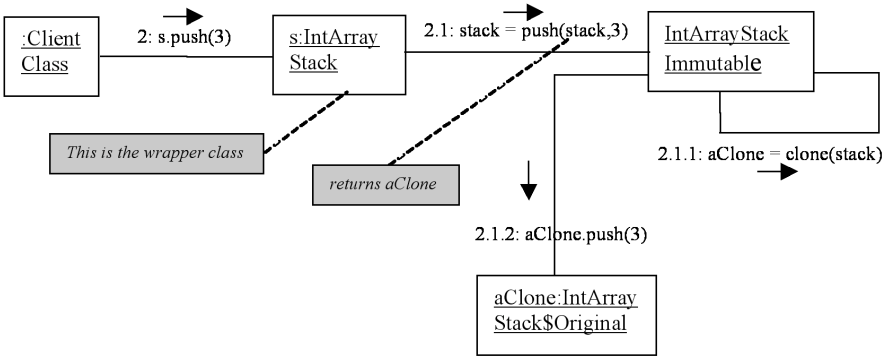

Fig. 9. Invoking a method upon an IntArrayStack.

- axioms that describe the result of a given derived operation/predicate opp_d on generic instances of the sort, generate post-conditions for the method that implements opp_d;
- axioms that pertain to sort equality generate post-conditions for the equals method.

A refinement mapping induces a straightforward translation of formulæ and terms into Java expressions. There are a few points of complexity however: (1) the translation of terms $op(t_1, \ldots, t_n)$ into method invocations; (2) the translation of strong equality used in axioms; (3) avoiding calls to methods, within contracts, in cases where their arguments are undefined.

In what concerns point (1), the return type of the method that implements *op* of specification *T* dictates the form of the translation: *(i)* if method m of class MyT that implements *op* is **void**, then $op(t_1, \ldots, t_n)$ is translated into an expression of the form MyT$Immutable.m(...); *(ii)* if method m of class MyT that implements *op* is not **void**, then a pair <value, state> must be returned by the MyT$Immutable version of method m, where value stands for the result of the method, and state stands for the target object state after m's invocation. $op(t_1, \ldots, t_n)$ is translated into an expression of the form MyT$Immutable.m(...).state if the sort of *op* is *T*, and MyT$Immutable.m(...).value in all other cases.

In what concerns point (2), the meaning of an equality $t_1 = t_2$ in the axioms of a specification is that the two terms are either both defined and have the same value, or they are both undefined. To be consistent with this definition, the evaluation of equals(t1, t2) within contracts should only be performed if t_1 and t_2 are both defined.

```
IntegerSpec is primitive int
  zero(x: Integer): Integer is 0;
  suc(x: Integer): Integer is x + 1;
  pred(x: Integer): Integer is x − 1;
  lt(x: Integer, y: Integer) is x < y;
ElemSpec is class String;
StackSpec is class StringArrayStack
  clear(s: Stack): Stack is void clear();
  push(s: Stack, e: Elem): Stack is void push(String e);
  pop(s: Stack): Stack is String pop();
  top(s: Stack): Elem is String top();
  size(s: Stack): Integer is int size();
  isEmpty(s: Stack) is boolean isEmpty();
```

Fig. 10. A refinement mapping for *GenericStack*.

```
public class StringArrayStack implements Cloneable {
  ...
  public void clear() { ... }
  public void push(String i) { ... }
  public String pop() { ... }
  public String top() { ... }
  public int size() { ... }
  public boolean isEmpty() { ... }
  public boolean equals (Object other) { ... }
  public Object clone() { ... }
}
```

Fig. 11. Java implementation of a String array stack.

If t_1 and t_2 are both undefined then the equality $t_1 = t_2$ is considered to hold and if just one of them is undefined, then the equality is false.

Finally, point (3) has to do with the fact that contracts of methods invoked within contracts are not monitored by the JML runtime assertion checker. Thus, we have to avoid making, in our contracts, method invocations with undefined arguments. A *def* function is defined and used in the translation process that supplies the definedness conditions for both terms and formulæ of our specification language (see [19] for the definition). As an example, the definedness condition for an operation call *op*$(t_1, \ldots, t_n)$ is the conjunction of the definedness conditions of terms t_1 to t_n with the domain condition of *op*.

We now present in more detail, for each type of axiom, the contract that result from applying the automatic translation process rules. A more complete description of the translation process can be found elsewhere [19]. We use $[\phi]$ to denote the translation of a formula ϕ, and *def*(ϕ) to denote the definedness condition for ϕ.

We illustrate the translation rules with a module *GenericStack*, with three components: the specification of integers presented in Figure 2 under the name *IntegerSpec*, a specification that simply declares the sort *Elem* under the name *ElemSpec* and a specification of stacks, under the name *StackSpec*, that only differs from the one in Figure 2 by the sort of its elements, which is the external sort *Elem* belonging to the imported specification *ElemSpec*. We choose the classes StringArrayStack and java.lang.String to refine module *GenericStack* through the refinement mapping of Figure 10.

Translation of Domain Restrictions. A domain restriction ϕ for an operation *op* generates a pre-condition for the method that implements *op*. In JML, pre-conditions are preceded by keyword **requires** and, thus, the pre-condition that results from translating the domain condition is **requires** $[def(\phi) \Rightarrow \phi]$.
Example: Domain restriction *top(s): **if** not isEmpty(s)* in specification *StackSpec*, Figure 2, produces pre-condition

```
requires true ==> !isEmpty(s).value;
```

in method String$Pair top(StringArrayStack$Original s) of class StringArrayStack$Immutable. The expression **true** is the definedness condition of variable s.
Translation of Axioms about Constructors and Observers. Axioms that specify the result of both constructors and observers on a given constructor op_c generate post-conditions for the method that implements op_c. In JML, post-conditions are preceded by keyword **ensures**. The post-conditions that result from translating axioms of the form $(\phi \Rightarrow op(op_c(\vec{x}), \vec{t}) = t)$ and $(\phi \Rightarrow pred_o(op_c(\vec{x}), \vec{t}))$ are

ensures $[def(\phi) \wedge \phi \Rightarrow op(r, \vec{t}) = t]$
ensures $[def(\phi) \wedge \phi \wedge def(pred_o(r, \vec{t})) \Rightarrow pred_o(r, \vec{t})]$

where *op* is a constructor or an observer operation, $pred_o$ is an observer predicate, and where r stands for the result of $op_c(\vec{x})$.
Example: Axiom *pop(push(s, i)) = s* produces post-condition

```
ensures true ==>                                    [φ] implies
  !(true && !isEmpty(\result).value) && !true ||    both undefined or
  (true && !isEmpty(\result).value) && true &&      (both defined and
    equals(pop(\result).state, s).value;             equal)
```

for method StringArrayStack$Original push (StringArrayStack$Original s, String i) in class StringArrayStack$Immutable. The expression **result** in JML represents the result of the method to which the post-condition is attached. The argument of *pop* in the axiom is *push(s, i)* which is precisely the result of method push. This post-condition is the translation of an equality between the terms *pop(push(s, i))* and *s*, thus it must evaluate to true if either both terms are undefined or they are both defined and have the same value. Remember that the definedness condition of an operation invocation is the conjunction of the definedness conditions of its arguments and the domain condition of the operation itself. The translation of the axiom *top(push(s, i)) = i* would be similar, except in the last part where we would have String$Immutable.equals(top(**result**).value, i).
Translating Axioms about Derived Operations/Predicates. Axioms that describe the result of a given derived operation/predicate opp_d on generic instances of the sort, generate post-conditions for the method that implements opp_d. The post-conditions that result from translating axioms $(\phi \Rightarrow op_d(\vec{x}) = t)$ and $(\phi \Rightarrow pred_d(\vec{x}))$ are

ensures $[def(\phi) \wedge \phi \Rightarrow r = t]$
ensures $[def(\phi) \wedge \phi \Rightarrow r]$

where op_d and $pred_d$ denote derived operations and predicates, and r stands for the result of $op_d(\vec{x})$ or $pred_d(\vec{x})$.
Example: Axiom *isEmpty(s) **if** size(s) = zero(_)* translates into the following post-condition in method boolean$Pair isEmpty (StringArrayStack$Original s).

```
ensures !(true && true) && !true ||                    both undefined or
  (true && true) && true &&                            (both defined and
  size(s).value == 0 ==>                                equal) imply
  \result.value                                        [r]
```

Translation of Axioms about Equality. Equality between values of a given type are regarded, to some extent, as type-specific predicates. In this way, axioms of the form $(\phi \Rightarrow x = x')$ generate post-conditions **ensures** $[def(\phi) \wedge \phi]$ ==> **result**.value for method equals. We do not illustrate this case since no specification in module *Generic-Stack* defines axioms of this kind.

Closing Assertions. Whenever the assertions (pre and post-conditions) contain a variable v that does not correspond to any of the parameters of the method to which the assertion belongs, the assertion must be preceded by a JML quantifier **forall** that quantifies over that variable within a given domain. Populating these domains is orthogonal to contract generation. In a technical report [19] we present a specific strategy for populating these domains—the one we have used for benchmarking our approach.

6.2 Enforcing Generic Properties

So far we have focused on the generation of contracts capturing user-defined properties, specific for a given type. In addition, there are generic properties concerning equality and cloning that are important to capture through contracts.

Contract for equals. In equational logic, any two terms that are regarded as equal must produce equal values for every operation and predicate. In order to check the consistency of an implementation in what respects these properties, our approach involves the automatic generation of post-conditions for the equals method that test the results given by all methods that implement observer operations and predicates when applied to the two objects being compared. More concretely, for every observer operation and predicate opp_o, the post-condition

```
ensures \result.value ==> (other instanceof C$Original &&
                           [opp_o(one, x⃗) = opp_o((C$Original) other, x⃗))]
```

is generated for the boolean$Pair equals(C$Original one, Object other) method of a class C$Immutable, where C is the class that implements the given specification.

The first part of the above expression is a Java boolean expression, while the second part denotes the translation of an equality between terms of our specification language extended with the (C) x term. The translation of this new term is itself, as expected. The translation of the term equality follows the rules previously explained.

Example: Returning to our StringArrayStack example, the contract generated for method boolean$Pair equals(StringArrayStack$Original one, Object other) in class StringArrayStack$Immutable includes the following pre-condition.

```
ensures \result.value ==> other instanceof StringArrayStack$Original &&
  (!!isEmpty(one).value &&
    (!!isEmpty((StringArrayStack$Original) other).value ||
    (true && !isEmpty(one).value && true) &&
    (true && !isEmpty((StringArrayStack$Original) other).value && true) &&
    String$Immutable.equals(top(one).value,
                            top((StringArrayStack$Original) other).value))
```

This contract does not completely capture congruence—it only tests observers applied to the left and right terms of equality. The process of testing equality between all terms obtained from the application of all combinations of observers is not realistic in this context. Instead, we rely on the not completely exhaustive process of monitoring, which heavily uses the equals method. Although the contracts of methods invoked from contracts are not monitored, we may force the execution of equals from within the immutable class equipped with contracts—this is a subject for further work.

We do not generate post-conditions for properties other than congruence, e.g. reflexivity and symmetry. Although these properties are crucial (as testified by the Java API contract for equals saying that it should implement an equivalence relation), given that they are independent of the target specification we chose not to enforce their checking—it would impose an important overhead.

Contracts for clone. Our approach makes use of cloning so, its soundness can be compromised if given implementations for clone do not meet the following correctness criteria: *(i)* the clone method is required not to have any effect whatsoever on **this**; *(ii)* clone's implementation is required to go deep enough in the structure of the object so that any references shared with the cloned object cannot get modified through the invocation of any of the remaining methods of the class. For example, an array based implementation of a stack, in which one of its methods changes the state of any of its elements, requires the elements of the stack to be cloned together with the array itself.

The post-condition that we want for method clone is one that imposes equality between the cloned object and the original one: **ensures** equals(**result**, o) is generated as a post-condition for the method Object clone(C$Original o) of class C$Immutable.

7 Congu

Congu [7] is a prototype that supports the approach by checking the consistency of specification modules and refinement mappings, and generating the classes required for monitorization. Given the user supplied entities—specification module, refinement mapping and classes—the following situations are identified as errors: the refinement mapping refers to an operation that is not present in the specification module; the refinement mapping refers to a method that is not present in the implementing classes; there are specification operations that are not mapped into any class method; among many others. Once contracts are generated and execution is monitored, the usual pre and post-condition exceptions are launched whenever invocations violate specification domain conditions, and operation implementations violate specification axioms.

We have tested our architecture on four data types: the stack specification described in this paper (both with *Stack* refined into an array-based "standard" class (Figure 3) and into java.util.Stack), a data type representing rational numbers, and a data structure *Vector* whose elements are indexed by integer values. The source code for the test cases can be found elsewhere [7]. For each data type we assessed the time and space used in five different situations.

1. The user's class and the test class only, both compiled with Sun's Java compiler, thus benchmarking the original user's class only;

2. The whole architecture compiled with Sun's Java compiler, thus benchmarking the overhead of our architecture, irrespective of the contracts;
3. The class responsible for checking the contracts, with its contracts removed, compiled with the JML compiler; all other classes in the architecture compiled with Sun's Java compiler.
4. As above but with all contracts in place, except that JML **forall** ranges were not generated;
5. As above but monitoring **forall** assertions with a limit of 20 elements in each range (see below).

All tests were conducted on a PC running Linux, equipped with a 1150 MHz CPU and 512Mb of RAM. We have used J2SE 1.4.2_09-b05 and JML 5.2. The runtime in seconds for 1.000.000 random operations, average of 10 runs, are as follows.

	Case 1	Case 2	Case 3	Case 4	Case 5	Slowdown
StringArrayStack	2.71	3.26	11.58	21.21	21.21	7.8
java.util.Stack	2.27	4.35	10.66	23.07	23.07	10.2
Rational	2.97	4.71	26.74	38.06	58.72	19.8
Vector	3.80	5.24	15.30	26.34	425.18	111.9

Inspecting the numbers for the first and the fifth case one concludes that monitoring introduces a 10 to 100-fold time penalty, depending on how many **forall** assertions are needed (none for the stacks, very little for the rational, a lot for the vector). The numbers for the second case indicate that conveying all calls to the data structure under testing through the Immutable class imposes a negligible overhead, when compiled with Sun's Java compiler. The numbers for the third case allow to conclude that roughly half of the total overhead reported in the fourth column is due to contract monitoring alone, while the other half to the fact that we are using the JML compiler. Comparing the fourth to the fifth case one concludes that monitoring **forall** assertions can impose quite an overhead, if the number of elements inside the **forall** domain is not properly limited. We omit the results on the space used; it suffices to say that the largest increase was reported for the *Vector* test, where we witnessed a negligible 5% increase from case 1 to case 5.

8 Related Work

There is a vast amount of work in the specification and checking of ADTs and software components in general; the interested reader may refer to previous publications [9, 12] for a survey. Here we focus on attempts to check OO implementations for conformance against property-driven algebraic specifications.

Henkel and Diwan developed a tool [13] that allows to check the behavioral equivalence between a Java class and its specification, during a particular run of a client application. This is achieved through the automatic generation of a prototype implementation for the specification which relies on term rewriting. The specification language that is adopted is, as in our approach, algebraic with equational axioms. The main difference is that their language is tailored to the specification of properties of OO implementations

```
forall l: LinkedList forall o: Object forall i: int
  removeLast(add(l, o).state).retval == o
  if i > 0 get(addFirst(l, o).state, intAdd(i, l).retval).retval == get(l,i).retval
```

```
axioms
  l: LinkedList; o: Elem, i: Integer;
  removeLast(add(_, o)) = o;
  get(AddFirst(l, o), suc(i)) = get (l, i) if gt(i, zero(_));
```

Fig. 12. An example of the specification of two properties of linked lists as they are presented by Henkel and Diwan [13] and as they would be specified in our approach.

whereas our language supports more abstract descriptions that are not specific to a particular programming paradigm. Being more abstract, we believe that our specifications are easier to write and understand.

Figure 12 presents an example. The axioms define that removeLast operation provides the last element that was added to the list and define the semantics of get operation: get(l, i) is the i-th element in the list l. The symbols retval and state are primitive constructs of the language adopted by Henkel and Diwan [13] to talk about the return value of an operation and the state of the current object after the operation, respectively.

When compared with our approach, another difference is that their language does not support the description of properties of operations that modify other objects, reachable from instance variables, nor does the tool. In contrast, our approach supports the monitoring of this kind of operation.

Another approach whose goal is similar to ours is Antoy and Hamlet's [1]. They propose an approach for checking the execution of an OO implementation against its algebraic specification, whose axioms are provided as executable rewrite rules. The user supplies the specification, an implementation class, and an explicit mapping from concrete data structures of the implementation to abstract values of the specification. A self-checking implementation is built that is the union of the implementation given by the implementer and an automatically generated direct implementation, together with some additional code to check their agreement. The abstraction mapping must be programmed by the user in the same language as the implementation class, and asks user knowledge about internal representation details. Here lies a difference between the two approaches: our refinement mapping needs only the interface information of implementing classes, and it is written in a very abstract language. Moreover, there are some axioms that are not accepted by their approach, due to the fact that they are used as rewrite rules; for example, equations like *insert (X, insert (Y, Z)) = insert (Y, insert (X,Z))* cannot be accepted as rewrite rules because they can be applied infinitely often. We further believe that the rich structure that our specifications present, together with the possibility to, through refinement mappings, map a same module into many different packages all implementing the same specification, is a positive point in our approach that we cannot devise in the above referred approaches.

9 Conclusion and Further Work

We described an approach for testing Java classes against specifications, using an algebraic, property-driven, approach to specifications as opposed to a model-driven one. We believe that the simplicity of property-driven specifications will encourage more software developers to use formal specifications. Therefore, we find it important to equip these approaches with tools similar to the ones currently available for model-driven ones.

Specifications define sorts, eventually referring to other sorts defined by other specifications, which they import. Specifications are nameless, so, the decision of which specification to choose to define a given sort, is made only at module composition time. This promotes reuse at the specification level.

Due to the abstract, implementation independent, nature of the specification language we adopted, it is easy to check different Java packages against the same specification module. This only requires the definition of appropriate refinement mappings. In the case of different classes that implement the same interface, refinement mappings can be reused as well.

Our approach has some limitations, some of these being *structural* in the sense that they are not solvable in any acceptable way while maintaining the overall structure:

- Self calls are not monitored. This limitation has a negligible impact if client classes call all the methods whose behavior needs to be tested.
- The approach is highly dependent on the quality of the clone methods, supplied by the user.
- Conformance checking ignores properties of specifications when implemented by primitive types. However, as described at the end of Section 5, our approach unveiled a problem we were not aware of, in a given module. This problem can only be overcome with client-invasive approaches.

We intend to investigate the best way to solve the problem of side-effects in contract monitoring due to changes in the state of method parameters—our approach does not cover this problem yet. Cloning all parameters in every call to a method in the generated immutable class—as we do for the target object—does not seem a plausible solution. We believe that methods that change the parameters' state do not appear very often in OO programming, except perhaps in implementations of the *Visitor* pattern and other similar situations. In our opinion a better solution would allow the user to explicitly indicate in the refinement mapping whether parameters are modified within methods (the default being that they are not modified).

The relation between domain conditions of specifications and exceptions raised by implementing methods is also a topic to investigate and develop, insofar as it would widen the universe of acceptable implementation classes.

We also plan to investigate possible extensions of both the specification and refinement languages in order to be possible to define 0-ary constructors, and refine them into Java constructors that override the default initialization.

A further topic for future work is the generation, from specifications and refinement mappings, of Java interfaces annotated with human readable contracts. Once one is convinced that given classes correctly implement a given module, it is important to make

this information available in the form of human-readable contracts to programmers that want to use these classes and need to know how to use and what they can expect from them.

Acknowledgments. This work was partially supported through the POSI/CHS/48015/2002 Project Contract Guided System Development project. Thanks are due to José Luiz Fiadeiro for many fruitful discussions that have helped putting the project together.

References

1. S. Antoy and R. Hamlet. Automatically checking an implementation against its formal specification. *IEEE TOSE*, 26(1):55–69, 2000.
2. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brckner, editors. *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports. Springer, 1999.
3. M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Proc. WSVCBS — OOPSLA 2001*, 2001.
4. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proc. of CASSIS 2004*, number 3362 in LNCS. Springer, 2004.
5. D. Bartetzko, C. Fisher, M. Moller, and H. Wehrheim. Jass - Java with assertions. *ENTCS*, 55(2), 2001.
6. M. Bidoit and P. Mosses. *CASL User Manual*. Number 2900 in LNCS. Springer, 2004.
7. Contract based system development. http://labmol.di.fc.ul.pt/congu/.
8. H. Ehrig and G. Mahr, editors. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, 1985.
9. J. Gannon, J. Purtilo, and M. Zelkowitz. Software specification: A comparison of formal methods, 2001.
10. J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In *Current Trends in Programming Methodology*, volume IV: Data Structuring, pages 80–149. Prentice-Hall, 1978.
11. J. Guttag, J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.
12. J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *Proc. ECOOP 2003*, LNCS, 2003.
13. J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *Proc. ICSE 2004*, 2004.
14. M. Huges and D. Stotts. Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In *Proc. ISSTV*, pages 53–61. ACM, 1996.
15. M. Karaorman, U. Holzle, and J. Bruno. jContractor: A reflective Java library to support design by contract. In *Proc. of Meta-Level Architectures and Reflection*, number 1616 in LNCS. Springer, 1999.
16. R. Kramer. iContract - The Java Design by Contract Tool. In *Proc. TOOLS USA'98*. IEEE Computer Society Press, 1999.
17. G. Leavens, K. Rustan, M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in java. In *OOPSLA'00 Companion*, pages 105–106. ACM Press, 2000.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall PTR, 2nd edition, 1997.
19. I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. Reis. Testing implementations of algebraic specifications with design-by-contract tools. DI/FCUL TR 05–22, 2005.
20. J. Spivey. *The Z Notation: A Reference Manual*. ISCS. Prentice-Hall, 1992.
21. Man Machine Systems. Design by contract for java using jmsassert. Published on the internet, 2000.