Testing Implementations of Algebraic Specifications with Design-by-Contract Tools

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

DI–FCUL TR–05–22

December 2005

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Testing Implementations of Algebraic Specifications with Design-by-Contract Tools

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

[2] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, PORTUGAL
`{in,mal,vv,joão.abreu,lmsar}@di.fc.ul.pt`

Abstract. We present an approach for testing Java implementations of abstract data types (ADTs) against their specifications. The key idea is to reduce this problem to the run-time monitoring of contract annotated classes, which is supported today by several runtime assertion-checking tools. The approach comprises an ADT specification language that allows automatic generation of monitorable contracts and a refinement language that supports the specification of the details of object-oriented implementations of ADTs.

1 Introduction

The definition of a data type in a way that is independent of its concrete representation on a specific programming language is a valuable asset in the design of algorithms and software systems. When reasoning at an abstract level, concrete data types that are equal up to renaming of data domains, data items and operations should not be distinguishable: we are only interested in their properties, not in their implementation. This leads to the notion of an abstract data type as proposed in the early 70's by Goguen et al [10].

Design by Contract (DBC) [21] is a software methodology based on the concept of abstract data type, integrating specification, design, and testing, and aiming at producing provably correct pieces of object-oriented software. In this approach, ADT specifications are class interfaces (Java interfaces, Eiffel abstract classes, etc) annotated with contracts expressed in a particular assertion language. Any implementation can be tested against its specification, by means of contract monitorization.

Property-driven algebraic approaches to ADT specification provide the conceptual basis for using data types in software design. Data types are defined by a set of sorts, a set of operations on those sorts and a set of axioms that support the precise specification of the semantic aspects. These specifications define classes of algebras, also called models. Such algebras can be regarded as possible implementations of the data types that are defined by the specification.

In purely model-driven approaches, data types are specified through a very abstract implementation based on primitive, but not necessarily basic, data types available in the adopted specification language. The abstract mapping that has to be supplied to describe the relation between the models and the structures that are chosen in the given implementation, can be rather difficult to obtain.

Among these approaches to ADT specification, the property-driven algebraic approach is the one that provides simpler and more concise specifications of ADTs and support specification at a higher level of abstraction.

The simplicity and expressive power of property-driven specifications can encourage more software developers to use formal specifications. Therefore, we find it important to equip property-driven approaches with tools similar to the ones currently available for model-driven approaches. Support for checking implementations against algebraic specifications is, as far as we know, restricted to the approaches proposed in [13, 1], which have some limitations.

In this paper, we propose an approach to system development that aims at bridging the gap between algebraic specifications and class specifications in an OO programming language, providing the means for software developers to benefit from the advantages of both worlds. The approach is tailored to Java and JML [20] but it could as well be defined towards other OO programming languages and assertion languages.

The key idea of the approach is to reduce the problem of testing Java implementations against specifications of ADTs to the run-time monitoring of contracts, which is supported today by many runtime assertion checking tools (e.g., [3, 17, 18, 19]). The ADTs' implementations we want to test become wrapped by other automatically generated classes. These are annotated with contracts that are generated from the corresponding ADTs specifications. The monitoring of these contracts at run-time is equivalent to the testing of the implementations against their specifications.

The paper is organised as follows. Section 2 motivates the use of a property-driven algebraic approach to the specification of ADTs by comparing it with DBC and model-driven approaches. Section 3 gives an overview of our approach, whose main components are presented in more detail in the subsequent sections. Section 4 presents the adopted algebraic specification language and the notion of refinement between specifications and Java classes. Section 5 describes the contracts that are generated from specifications. Section 6 presents benchmarking results obtained from testing our architecture on three data types. Section 7 presents some related work, and section 8 concludes and describes some topics that need further work.

2 Motivation

In order to motivate and illustrate our approach we use the abstract data type *integer stack*. Figure 1 presents a specification of this data type. We follow the usual style of property-driven algebraic specifications [8]. In general terms, we adopt partial specifications with conditional axioms.

```
specification
    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: IntStack
              top(s),pop(s): if ¬isEmpty(s)
    axioms    s:IntStack, i:Integer
              top(push(_,i)) = i                    //Ax1
              pop(push(s,_)) = s                    //Ax2
              size(clear(_)) = zero(_)              //Ax3
              size(push(_,s)) = suc(size(s))        //Ax4
              isEmpty(s) ⇐ size(s)= zero(_)         //Ax5
              ¬isEmpty(s) ⇐ ¬(size(s)= zero(_))     //Ax6
    end
```

Fig. 1. Specification of an integer stack.

Operation symbols are classified in one of three classes:

- *constructors*, representing the operations with which all the values of the type can be obtained;
- *observers*, representing the operations that give fundamental information about the values of the type;
- *derived*, which represent all other operations.

Operation symbols declared with ⇒? can be interpreted by partial functions. The section *domains* of the specification, allows to describe the conditions under which interpretations of these operations are required to be defined.

To achieve software reliability, given a Java implementation of integer stacks – for instance, the class *ArrayStack* presented in Figure 2 – it is important that this implementation can be checked against the specification.

A well-known approach to this problem is to adopt DBC. In this case, the semantic properties of the data type are specified through assertions – pre-conditions, post-conditions and invariants – that can be monitored at run-time. Pre-conditions can be used to express domain conditions of operations: if they are monitored, no client succeeds in executing an operation in a state that does not belong to the operation domain. Post-conditions can be used to express the other semantic properties of the operations: they are often expressed by relating the objects as they were before operation execution with the objects as they are after it. If assertions are monitored, an exception is usually raised whenever a situation is found in which the implementation does not conform to the specification.

```
Java class

public class ArrayStack 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;
           size++;
      }
      public void pop (){
           size--;
      }
      public int top (){
           return elems[size-1];
      }
      public int size (){
           return size;
      }
      public boolean isEmpty (){
           return size==0;
      }
      private void reallocate (){
           ....;
      }
      public boolean equals (Object other){
           ....;
      }
      public Object clone (){
           ....;
      }
}
```

Fig. 2. Java implementation of an integer stack.

Let us now analyse the support given by DBC to the specification of the data type *integer stack* through the integration of assertions in the class *ArrayStack*. Following B.Meyer [21], this could be achieved by adding contracts as shown in Figure 3.

The conditions defined under the *domains* section of the ADT specification (cf. Figure 1) are used here as pre-conditions. The post-conditions capture the functionality of the operations specified by the axioms. Notice that there are axioms that can not be captured by any conceivable post-condition, due to the impossibility of expressing them in terms of *monitorable* post-conditions. This is the case of axiom 2 – *pop(push(s,_)) = s*. The inclusion of a post-condition in method *push* with the flavour of *pop().equals(\old clone( ))* would not work because *pop* is a *void* method.

The example shows that, whenever a specification is implemented by a mutable type, there are axioms that may not be expressible as monitorable contracts of the class.

```
Java class

public class ArrayStack implements Cloneable{
         ...
      /*@ ensures  size() == 0; */
      public void clear () {… }

      /*@ ensures top() == i && size() == \old size() + 1; */
      public void push (int i) {… }

      /*@ requires !isEmpty(); */
      /*@ ensures  size() == \old (size() - 1) */
      public void pop () {… }

      /*@ requires !isEmpty(); */
      public /*@ pure @*/ int top () {… }

      public /*@ pure @*/ int size() {… }

      /*@ ensures  \result == (size() == 0); */
      public /*@ pure @*/ boolean isEmpty() {… }
         ...
}
```

Fig. 3. Adding contracts to *ArrayStack*.

Also notice that we have classified *size* as an observer operation and *isEmpty* as derived. Had we done otherwise (a more natural choice) we would be left with the axioms:

```
size(s)= zero(_)             ⇐  isEmpty(s)                 \\ Ax 7
size(s)= suc(size(pop(s)))   ⇐  ¬isEmpty(s)                \\ Ax 8
```

and would not be able to express these properties as post-conditions in method *size*, again because *pop* is a *void* method.

These problems disallow the expression of important properties of many methods in common data types. Unless we have methods that allow to inspect the whole structure of the data type elements without modifying it (for instance a method *element(i)* for inspecting every *i-th* element of the stack), we are not capable of writing complete monitorable post-conditions. These inspection methods are, in general, artificial, and even against the nature of the type itself and, hence, they are not a solution to the problem.

As a result, we cannot directly rely on DBC for monitoring property driven ADT specifications.

Purely model-oriented approaches, like the ones followed by users of Z [24], Larch [11], JML [20], etc, are another alternative to the specification of abstract data types and the checking of implementations against specifications. These approaches support the description of abstract implementations – defined, for example, in terms of sets or lists – which are then used as abstract models of the types under specification.

Figure 4 presents an example of a model-based specification of stacks taken from [22]. This specific model of stacks relies on sequences, more concretely on objects of type *JMLObjectSequence* – a class belonging to the distribution of JML. The class *JMLObjectSequence* defines immutable sequences, including a series of methods for sequence manipulation from which the methods *trailer()*, *insertFront()*, *first(),* that are used in this specification, are examples. The model underlying *JMLObjectSequence* is a finite sequence of elements.

Java class

```
//@ model import org.jmlspecs.models.*;
public abstract class UnboundedStack {
  /*@ public model JMLObjectSequence theStack;
    @ public initially theStack != null && theStack.isEmpty();
    @*/
  //@ public invariant theStack != null;
  /*@ public normal_behavior
    @   requires !theStack.isEmpty();
    @   assignable theStack;
    @   ensures theStack.equals(\old(theStack.trailer()));
    @*/
  public abstract void pop( );
  /*@ public normal_behavior
    @   assignable theStack;
    @   ensures theStack.equals(\old(theStack.insertFront(x)));
    @*/
  public abstract void push(Object x);
  /*@ public normal_behavior
    @   requires !theStack.isEmpty();
    @   assignable \nothing;
    @   ensures \result == theStack.first();
    @*/
  public /*@ pure @*/ abstract Object top( );
}
```

Fig. 4. JML specification of *UnboundedStack*.

When a specific implementation of *UnboundedStack* is defined, it is necessary to explicitly describe the relation between the *JMLObjectSequence theStack* and the structure that is chosen to store the stack elements. This relation is known as the *abstraction function*. Figure 5 partially illustrates the definition of this relation for an implementation of stacks as *ArrayList*s as presented in [22].

Although we recognise the important role played by model-based approaches to ADT specification, we think that, for a significant part of software developers, it can be rather difficult to write this kind of specifications. We believe that the simplicity of property-driven specifications can encourage more software developers to use formal specifications. Therefore, we find it important to equip property-driven approaches with tools similar to the ones currently available for model-driven approaches.

Java class

```
// Partial implementation of UnboundedStack
public class UnboundedStackAsArrayList extends UnboundedStack {

  protected ArrayList elems;
  //@                     in theStack;
  //@                     maps elems.theList \into theStack;

  /*@ protected represents theStack <- abstractValue();
   @  protected represents_redundantly theStack \such_that
   @     (\forall int i;
   @         0 <= i && i < elems.size();
   @         elems.get(i) == theStack.itemAt(i) );
   @*/

  /*@ protected pure model JMLObjectSequence abstractValue() {
    @   JMLObjectSequence ret = new JMLObjectSequence();
    @   Iterator iter = elems.iterator();
    @   while (iter.hasNext()) {
    @     ret = ret.insertBack(iter.next());
    @   }
    @   return ret;
    @ }
    @*/
…
}
```

Fig. 5. Partial view of an implementation of *UnboundedStack*.

3 Approach Overview

In this section we provide an overview of our approach for testing Java implementations of ADTs against property-driven specifications. For the purpose of this overview, we consider a simplified scenario in which the implementation that we want to test against a given specification module is composed of one only class – this does not imply that the specification refers to one only sort, but that it is mapped to one class and, possibly, some primitive types.

The approach is depicted in Figure 6. From a user-centric point of view, the approach includes (1) a specification *T* – the specification of the ADT, (2) a Java class *MyT* – the implementation that we want to test and (3) a refinement mapping between *T* and *MyT* – the definition of the relationship between the operation symbols of the specification and the method names of the implementation, among other things.

As a result, two classes are produced – a wrapper of *MyT* class and an immutable *MyT$Immutable* class – that ensure that, during the execution of a system involving classes that are clients of the original *MyT* class: a) the behaviour of the original *MyT* objects will be checked against specification *T*, and b) the clients' behaviour with respect to the original *MyT* operations will be monitored.

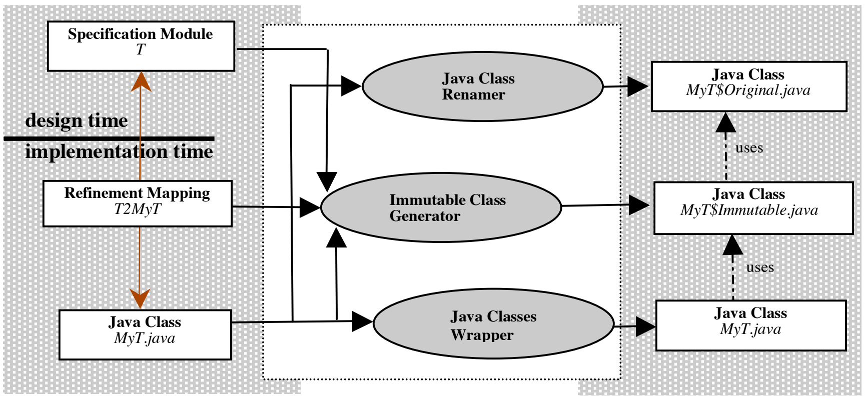


Fig. 6. Approach overview.

The generated *MyT* wrapper 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. This 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$Immutable* – in response to client calls. Because the class *MyT$Immutable* is annotated with contracts, the properties of the ADT are checked (and the violations are reported), whenever the system is executed under the observation of a contract monitoring tool. See Figures 7 and 8 for an example.

In order to avoid modifying *MyT* clients so as to become clients of the wrapper class instead, the original *MyT* class is renamed – its name is postfixed with "*$Original*". In this way, the monitoring of *MyT* original class becomes transparent to client classes. Let *m* be a method of the original *MyT* class. The corresponding method *m* in the wrapper class *MyT* calls the immutable version of *m* in *MyT$Immutable*, using its wrapped attribute (whose type is *MyT$Original*) as the first argument of the call. This immutable version of method *m* calls the original *m* over a clone of the *MyT$Original* argument, and returns its result. Due to the fact that it does not modify its *MyT$Original* argument, while returning the desired object, the immutable version of *m* can be invoked in contracts of its own class *MyT$Immutable*, allowing to test the effects of the original *m*.

Concerning Figures 7 and 8, since *clear()* is a *void* method, then the corresponding method in *ArrayStack$Immutable* must return the *ArrayStack$Original* object that results from applying *clear* to a clone of the original object. The result of this call is then stored in the *stack* attribute.

In the case of a non-*void* method, say *size*, the corresponding *size* method in *ArrayStack$Immutable* returns a pair <*value,state*> where *value* stands for the result of the method, and *state* stands for the target object state after *size*'s invocation. The wrapper class – the new *ArrayStack* – uses the *value* part of this pair to return the value to the client, and the *state* part of this pair to assign it to its only attribute (in order to account for methods that, in addition to returning a value, also modify the current object).

```
Java class  (i)

// Automatically generated from the original ArrayStack class,
// now renamed to ArrayStack$Original
public class ArrayStack implements Cloneable {
       protected ArrayStack$Original stack = new ArrayStack$Original ();
       public void clear() {
         stack = ArrayStack$Immutable.clear(stack);
       }
       public void push(int i) {
         stack = ArrayStack$Immutable.push(stack, i);
       }
       public void pop() {
         stack = ArrayStack$Immutable.pop(stack);
       }
       public int size() {
         int$Pair result = ArrayStack$Immutable.size(stack);
         stack = result.state();
         return result.value());
       }
       public int top() {
         int$Pair result = ArrayStack$Immutable.top(stack);
         stack = result.state();
         return result.value());
       }
       ... // this class is not complete
}
```

```
Java class  (ii)

public class int$Pair {
     private final int value; ArrayStack$Original state;
     public int$Pair (int value, ArrayStack$Original state){
        this.value = value;
        this.state = sstate;
     }
     public int value() {
        return value;
     }
     public ArrayStack$Original state() {
        return state;
     }
}
```

Fig. 7. **(i)** Partial view of the wrapper class that results from applying our approach to the specification *IntStackSpec* and the original *ArrayStack* class. **(ii)** The auxiliary class *int$Pair*.

To keep things simple, we have not presented the refinement mapping (cf. Figure 6) for this example; it will be fully explained in the next section. The role of the refinement mapping is to map each sort to a Java class or primitive type, and each ADT operator to the Java method or primitive operation that implements it.

Java class

```
// Automatically generated from the ArrayStack class and
// the IntStackSpec specification
public class ArrayStack$Immutable {
     /*@
      @ ensures size(\result).value()== 0;                   //Ax3
      @*/
     static public ArrayStack$Original clear (ArrayStack$Original s) {
         ArrayStack$Original result = (ArrayStack$Original) clone(s);
         result.clear();
         return result;
     }
    /*@
     @ ensures size(\result).value() == size(s).value() + 1; //Ax4
     @ ensures top(\result).value() == i;                    //Ax1
     @ ensures equals(pop(\result).state(), s).value();      //Ax2
     @*/
     static public ArrayStack$Original push (ArrayStack$Original s, int i) {
         ArrayStack$Original result = (ArrayStack$Original) clone(s);
         result.push(i);
         return result;
     }

    /*@
     @ requires !isEmpty(s).value();
     @*/
     static public ArrayStack$Original pop (ArrayStack$Original s) {
         ArrayStack$Original result = (ArrayStack$Original) clone(s);
         result.pop();
         return result;
     }

      static public int$Pair size (ArrayStack$Original s) {
         ArrayStack$Original clone = (ArrayStack$Original) clone(s);
         return new int$Pair (clone.size(), clone);
      }

     /*@
      @ensures <see Section 5.2>;
      @*/
      static public boolean$Pair equals (ArrayStack$Original one, Object other){
         ArrayStack$Original clone = (ArrayStack$Original) clone(s);
         return new boolean$Pair (clone.equals(other), clone);

      }

      ... //this class is not complete
}
```

Fig. 8. Partial view of the *Immutable* class that results from applying our approach to the specification *IntStackSpec* and the original *ArrayStack* class.

4 Specification Languages: ADTs and Refinements

As illustrated in the previous sections, we define abstract data types in terms of algebraic specifications. The language we adopt is, to some extent, similar to many existing languages, e.g. CASL [8]. It has, however, some specific features, such as the classification of operations in different categories, and strong restrictions on the form of the axioms.

Our teaching experience shows us that the imposition of rules to guide the task of specifying ADTs is useful. The rules we impose are not only intuitive and easy to understand and to apply, but they are also necessary in driving the automatic identification of contracts for classes.

The language provides for the description of *modules*. The building blocks it relies upon are *specifications*. A specification includes a set of symbol declarations – a sort s, a set of operations Ω, and a set of predicates P – and a set of axioms Φ. The declaration of operations and predicates includes the definition of their profile – the number and the sorts of the arguments. In the case of operations, the profile also includes the sort of the result and whether the operation is required to be total. The first argument of every operation or predicate must be of sort s (the sort that the specification defines); the reason for this requirement is discussed below.

Operations are classified as *constructors*, *observers* or *derived*, whereas predicates can only be classified as either *observers* or *derived*. Constructors represent the operations from which all the values of type s can be obtained. Observers represent the operations and predicates that give fundamental information about the values of type s. The same applies to operation and predicate symbols declared as derived, but in this case the provided information could have been obtained through the other operations and predicates. The classification of an operation/predicate as derived rather than observer, reflects itself on the form of the axioms, as well as on the Java method where we place the contract generated from the axiom.

The set of axioms is divided into two parts. The first part concerns the domain of definition of the operations – it defines conditions under which the interpretation of each operation must be defined. For operations declared as total (featuring $\Rightarrow$), the domain condition is implicit and, hence, does not need to be specified. If, for a given partial operation (specified with $\Rightarrow$?), no condition is indicated, it means that the operation can be interpreted by any function, including the one which is undefined everywhere.

The second part of the axioms expresses constraints over the interpretation of operation and predicate symbols. We identify four essential classes of axioms: (a) axioms that relate constructors; (b) axioms that specify the result of observers on constructors; (c) axioms that describe the result of derived operations/predicates on generic instances of the sort; (d) axioms that pertain to sort equality.

Axioms are closed formulae of first-order logic restricted to one of the following forms.

$$\forall \vec{y}\ (\ \phi \Rightarrow opC'(opC(\vec{x}),\vec{t}\ )=t\ ) \qquad (a)$$

$$\forall \vec{y}\,(\ \phi \Rightarrow opO(opC(\vec{x}),\vec{t}\ )=t\ ) \qquad (b)$$

$$\forall \vec{y}\,(\,\phi \Rightarrow predO(opC(\vec{x}),\vec{t}\,)\,) \qquad (b)$$
$$\forall \vec{y}\,(\,\phi \Rightarrow \neg predO(opC(\vec{x}),\vec{t}\,)\,) \qquad (b)$$
$$\forall \vec{y}\,(\,\phi \Rightarrow opD(\vec{x})=t\,) \qquad (c)$$
$$\forall \vec{y}\,(\,\phi \Rightarrow predD(\vec{x})\,) \qquad (c)$$
$$\forall \vec{y}\,(\,\phi \Rightarrow \neg predD(\vec{x})\,) \qquad (c)$$
$$\forall \vec{y}\,(\,\phi \Rightarrow x_1=x_2\,) \qquad (d)$$

where $\vec{y}$ is a list of variables, ϕ is a quantifier-free formula over $\vec{y}$, $\vec{x}$ is a list of variables in $\vec{y}$, x_1 and x_2 are variables in $\vec{y}$, $\vec{t}$ is a list of terms over $\vec{y}$, t is a term over $\vec{y}$, *opC* and *opC'*, *opO* and *opD* are operation symbols in Ω (Constructors, Observers and Derived operations, respectively), *predO* and *predD* are predicate symbols in P (Observers and Derived predicates, respectively).

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

To ease the reading, the concrete syntax used in the examples allows some simplifications. For instance, universal quantifiers are omitted, implications may be written from right to left and the symbol '_' is used, as in Prolog, for variables whose identity is irrelevant. The complete definition of the syntax of the language is the subject of a separate publication [15].

4.1 Specifications and Modules

A *specification* consists of a set of references to other specifications (*import-specs*, for short), a sort, a set of operations, a set of predicates and a set of axioms. The symbols used in the specification that are not locally declared (sorts, operations or predicates) are designated by *external symbols.* The specification of integer stacks presented in Figure 1 imports a specification – *IntegerSpec* – and uses sort *Integer* and operation symbols *zero* and *suc* which are external symbols.

A *module* puts together specifications while assigning them a name. When a specification is embedded, as a component, in a module, components must also be embedded that provide the specification's external symbols.

Whenever the set of *import-specs* is empty in a specification, we call it is a *closed* specification; these are essentially self-contained specifications with a single sort, which allow the specification of basic ADTs such as integers and booleans (cf. Figure 9).

A *module* is a surjective function $\mu{:}N \rightarrow \Xi$ from a set N (of specification names) to a set Ξ of specifications s.t. (1) the import-specs of specifications in Ξ are included in N and (2) every external symbol x of a specification in Ξ is declared exactly in one of the imported specifications – we use x^{μ} to denote the name of this specification.

In the case of our running example, in order to put together the specification of integers (Figure 9), and integer stacks (Figure 1), we have to assign names – for example, *IntegerSpec* and *IntStackSpec* – to both specifications (Figure 10).

```
specification
    sort Integer
    operations and predicates
            constructors
                    zero: Integer ⇒ Integer
                    suc: Integer ⇒ Integer
                    pred: Integer ⇒ Integer
            observers
                    _<_: Integer Integer
    domains

    axioms i,j:Integer
            zero(_) < suc(zero(_))                              //Ax1
            pred(zero(_)) < zero(_)                             //Ax2
            pred(i) < j ⇐ i<j                                   //Ax3
            zero(_) < suc(j) ⇐ zero(_)<j                        //Ax4
            suc(i) < suc(j) ⇐ i<j                               //Ax5
            pred(i) < pred(j) ⇐ i<j                             //Ax6
            pred(suc(i)) = i                                    //Ax7
            suc(pred(i)) = i                                    //Ax8
    end
```

Fig. 9. An example of a closed specification.

```
module
IntStackSpec      ⇛      Fig. 1

IntegerSpec       ⇛      Fig. 9
```

Fig. 10. *IntegerStack*: an example of a module.

4.2 Refinement Mappings

In order to check Java classes against specifications, it is necessary to provide mappings that bridge the gap between the two worlds. In our approach, the gap between modules and collections of Java classes is defined in terms of what we have called *refinement mappings* (cf. Figure 6). Before defining the mapping, we discuss some key aspects of OO implementations for data types.

In the context of the OO paradigm, a data type *t* is usually implemented by a class *T* whose objects are the values of *t*. Furthermore, operations and predicates of *t* are usually implemented as instance methods of *T*. This means that, whenever a client invokes a method of the class, it must provide the target object separately from the method arguments. Therefore, a refinement mapping must bind every *n+1*-ary operation or predicate of the data type to an *n*-ary method of *T*. The first argument of the operation is implicitly provided – it is the *target* object (remember that the sort of the first argument in all operations/predicates of any specification defining sort *s* must exist and be of sort *s*). In what concerns the return type, an operation whose result type is *t* can be either bound to a procedure (*void* method) or to a function of either type T or some T'. The former case is typical of mutable implementations, in which an object may represent different data values during its life time. This is also the case when the return type is T' different from T (for example, a method *int popTop* that implements operation *pop* and also returns the top element). Predicates have to be bound to methods of type *boolean*.

Although less common, elementary data types can also be implemented by (Java) primitive types. Our approach supports this form of implementation for closed specifications. In this case, the refinement mapping has to define the way operations and predicates of the data type are expressed in terms of built-in Java operations.

Refinement mappings: Given a module $\mu: N \to \Xi$ and a set $\mathcal{C}$ of Java types, a refinement mapping $\rho: N \to \mathcal{C} \times B$ is a N-indexed set of pairs $\{<C_v,\beta_v>\}_{v:N}$ where C_v is a type in $\mathcal{C}$ (which can be a primitive type only if $\mu(v)$ is a closed specification) and β_v is a binding between the specification $\mu(v)$ and the class C_v.
The refinement mapping ρ is such that, if $\rho(v) = < C_v,\beta_v>$,

and C_v is **primitive**, then

$\mu(v)$ is a closed specification, and

$\beta_v(op)$ is a Java expression of type C_v, and $\beta_v(pred)$ is a Java expression of *boolean* type, built for the set of variables $\{x_1,\ldots,x_n\}$, where n is the arity of *op* or *pred*.

and C_v is a **class**, then

$\beta_v(opp)$, where *opp* is either a predicate *pred* or an operation *op*, is a method signature in C_v such that,

(**arity**) β_v maps n+1-ary operations and predicates into n-ary methods;

(**return type**) $\beta_v(pred)$ is of *boolean* type; where t is the return type of *op*, $\beta_v(op)$ is of type of the class in pair $\rho(t^{\mu})$;

(**parameter type**) the i-th parameter of $\beta_v(opp)$ has the type of the class in pair $\rho(t^{\mu})$ where t is the sort of the $(i+1)$-th parameter of *opp*.

Notice that operations with result sort s can be mapped into methods with any return type, *void* included. This allows us to establish refinement mappings to classes with methods that achieve their expected functionality by changing the state of the current object and that additionally return some value (this is, for instance, the case of methods *add* and *remove* of the class *java.util.Collection* that return a *boolean* value indicating whether the execution of the method implied a change to the collection).

To illustrate these notions, consider again the module *IntegerStack* presented in Figure 10. In order to check the conformance of the Java class *ArrayStack* (Figure 2) against this specification we have to define an appropriate refinement mapping. An admissible choice is presented in Figure 11. It states that the Java primitive type *int* was chosen to implement the data type *IntegerSpec* and defines, for each of its operation and predicate symbols, the corresponding Java expression.

```
refinement mapping

IntegerSpec is primitive int
        zero(x1:Integer)                                is 0
        suc(x1:Integer):Integer                         is x1 + 1
        pred(x1:Integer):Integer                        is x1 - 1
        _<_(x1:Integer, x2:Integer)                     is x1 < x2
IntStackSpec is class ArrayStack
        clear(s:IntStack): IntStack                     is void clear()
        push(s:IntStack,i:Elem): IntStack               is void push(int i)
        pop(s:IntStack):IntStack                        is void pop()
        top(s:IntStack):Integer                         is int top()
        size(s:IntStack):Integer                        is int size()
        isEmpty(s:IntStack)                             is boolean isEmpty()
```

Fig. 11. An example of a refinement mapping.

Notice that a refinement mapping may define a mapping that maps two different specifications into the same class or primitive type. This is extremely useful since it promotes the writing of generic specifications that can be reused in different situations. As an example, consider the specifications presented in Figures 12a) and b).

```
specification  (a)

   import   IntegerSpec
   import   ElemSpec
   sort   Stack
   operations and predicates
       constructors
          clear: Stack ⇒ Stack
          push: Stack Elem ⇒ Stack
       observers
          top: Stack ⇒? Elem
          pop: Stack ⇒? Stack
          size: Stack ⇒ Integer
       derived
          isEmpty: Stack
       domains   s: Stack
                   …
       axioms  s:Stack, i:Elem,
                   …
end
```

```
specification (b)

    sort Elem
end
```

Fig. 12. a) The specification of a generic stack; **b)** a closed specification of general elements

Figure 13 shows a module – *GenerickStack* – that includes the above specifications, and also *IntegerSpec* of Figure 9.

```
module GenericStack

StackSpec        ⇛      Fig. 12a)

IntegerSpec      ⇛      Fig. 9

ElemSpec         ⇛      Fig. 12b)
```

Fig. 13. The module *GenericStack*

It is possible to bind both specifications *IntegerSpec* and *ElemSpec* to the primitive type *int* as explained previously and bind the specification *StackSpec* to the Java class *ArrayStack* of Figure 3. The example shows that our approach allows checking the implementation of *ArrayStack* against either module *IntegerStack* or module *Generic-Stack* by simply considering different refinement mappings.

5 Contract Generation

Given a module, a set of Java classes and primitive types implementing the specifications in the module, and a refinement mapping (between the specifications and the implementations), we generate several classes, some of them annotated with contracts (cf. Figure 6). In this section, we present the main rules that govern contract generation.

Given a module $\mu: N \to \Xi$, a set $\mathcal{C}$ of Java types that implements μ, and a refinement mapping $\rho = \{<C_v,\beta_v>\}_{v:N}$, we define how the axioms of specification $\mu(v)$ translate into assertions that constitute contracts for the methods of the class C_v*Immutable* where C_v is such that $\rho(v) = < C_v,\beta_v>$. Whenever a specification is implemented by a primitive type, no contract generation is achieved – it is not possible to attach pre and post-conditions to operations of primitive types.

Contract generation for the methods of C_v*Immutable* can be described in two parts: translation of explicit properties (axiom translation in Section 5.1), and translation of implicit properties (enforcing equality properties of equational logic in Section 5.2).

The generation of contracts that capture the properties explicitly specified in a given specification $\mu(v)$, is such that

- a domain restriction for an operation *op* generates a pre-condition for the method $\beta_v(op)$;
- an axiom of one of the forms

  $\forall \vec{y}\ (\ \phi \Rightarrow opC'(opC(\vec{x}),\vec{t}\ )=t\ )$

  $\forall \vec{y}\ (\ \phi \Rightarrow opO(opC(\vec{x}),\vec{t}\ )=t\ )$

  $\forall \vec{y}\ (\ \phi \Rightarrow predO(opC(\vec{x}),\vec{t}\ )\ )$

  $\forall \vec{y}\ (\ \phi \Rightarrow \neg predO(opC(\vec{x}),\vec{t}\ )\ )$

  generates a post-condition for method $\beta_v(opC)$;
- an axiom of the form

  $\forall \vec{y}\ (\ \phi \Rightarrow opD(\vec{x})=t\ )$

  generates a post-condition for method $\beta_v(opD)$;
- an axiom of one of the forms

  $\forall \vec{y}\ (\ \phi \Rightarrow predD(\vec{x})\ )$

  $\forall \vec{y}\ (\ \phi \Rightarrow \neg predD(\vec{x})\ )$

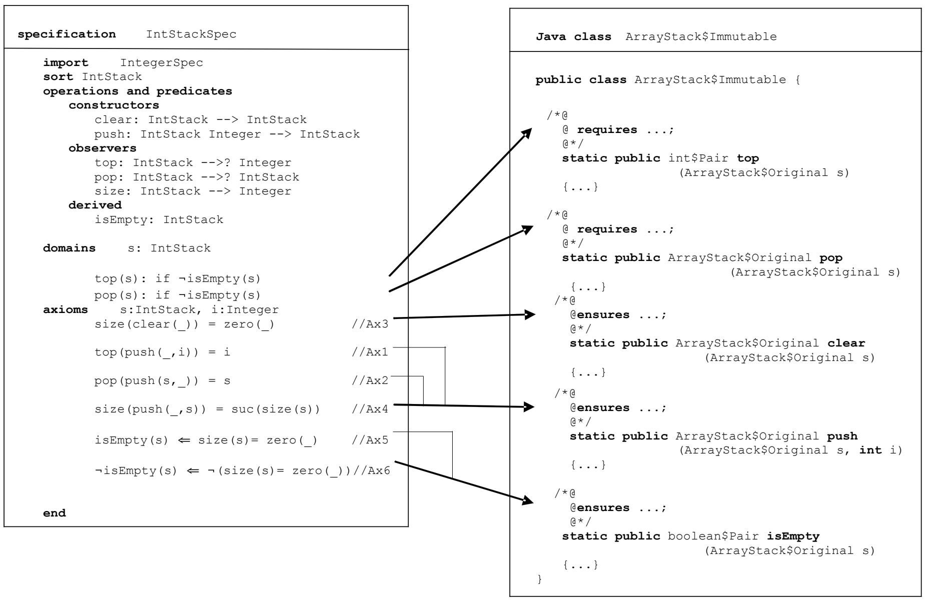
  generates a post-condition for method $\beta_v(predD);$

- an axiom of the form

  $\forall \vec{y}\ (\ \phi \Rightarrow x_1 = x_2\ )$

  generates a post-condition for method *equals*.

Consider, for instance, the refinement mapping defined in Figure 11. The following figure shows the destination methods, in class *ArrayStack$Immutable* (Figure 8), for the contracts that are generated from the axioms in *IntStackSpec* (Figure 1) specification.



```
specification    IntStackSpec

    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: IntStack

            top(s): if ¬isEmpty(s)
            pop(s): if ¬isEmpty(s)
    axioms    s:IntStack, i:Integer
            size(clear(_)) = zero(_)          //Ax3

            top(push(_,i)) = i                //Ax1

            pop(push(s,_)) = s                //Ax2

            size(push(_,s)) = suc(size(s))    //Ax4

            isEmpty(s) ⇐ size(s)= zero(_)     //Ax5

            ¬isEmpty(s) ⇐ ¬(size(s)= zero(_))//Ax6


    end
```

```
Java class  ArrayStack$Immutable

public class ArrayStack$Immutable {

  /*@
    @ requires ...;
    @*/
    static public int$Pair top
                         (ArrayStack$Original s)
    {...}

  /*@
    @ requires ...;
    @*/
    static public ArrayStack$Original pop
                               (ArrayStack$Original s)
     {...}
   /*@
     @ensures ...;
     @*/
     static public ArrayStack$Original clear
                            (ArrayStack$Original s)
     {...}
   /*@
     @ensures ...;
     @*/
     static public ArrayStack$Original push
                         (ArrayStack$Original s, int i)
     {...}

   /*@
     @ensures ...;
     @*/
    static public boolean$Pair isEmpty
                            (ArrayStack$Original s)
    {...}
}
```

Fig. 14. Destination methods for contracts generated from axioms

The second part of contract generation is concerned with properties of equational logic, namely the properties of the form

$$\forall \vec{y}\ (\ (x_1 = x_2) \Rightarrow op(\vec{x}_1) = op(\vec{x}_2)\ )$$

$$\forall \vec{y}\ (\ (x_1 = x_2) \Rightarrow pred(\vec{x}_1) \Leftrightarrow pred(\vec{x}_2)\ )$$

These properties are captured through the generation of post-conditions for the static method *equals()* of *C$_\surd$Immutable* class, for each observer operation and predicate of $\mu(v)$.

In the rest of this section we present in more detail the main rules that govern contract generation and illustrate them through pieces of the example presented in full in appendix A.

5.1 From Axioms to Contracts

We illustrate the translation rules with the module *GenericStack* presented in Figure 13 and the refinement mapping presented in Figure 16, where *StringStack* is the class presented in Figure 15.

```
Java class

public class StringStack 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. 15. Partial view of the class that implements *StackSpec* as provided by the user.

```
Refinement Mapping

IntegerSpec is primitive int
     zero(x1:Integer)                  is 0
     suc(x1:Integer):Integer           is x1 + 1
     pred(x1:Integer):Integer          is x1 - 1
     _<_(x1:Integer, x2:Integer)       is x1 < x2

ElemSpec is class java.lang.String

StackSpec is class StringStack
     clear(s:Stack):Stack              is void clear()
     push(s:Stack,i:Elem):Stack        is void push(java.lang.String i)
     pop(s:Stack):Stack                is java.lang.String pop()
     top(s:Stack):Integer              is java.lang.String top()
     size(s:Stack):Integer             is int size()
     isEmpty(s:Stack)                  is boolean isEmpty()
```

Fig. 16. A refinement mapping for module *GenericStack* in Figure 13.

Translation of Terms and Formulae

A refinement mapping induces a straightforward translation of formulas and terms into Java expressions. There are a few points of complexity, including the translation of terms *op*(t_1,…,t_n) into method invocations whose form depends on the return type of the method that implements *op*. This happens because the effect of the application of an operation can be achieved, in Java, either through a method that modifies the current object without returning a value, or through a method that returns a value (and that may or may not modify the current object).

The simplest case is that in which operation *op* is implemented through a *void* method *m* (of class *MyT*). This means that the effects of *op* are captured by the current object after the invocation of *m*. This is precisely the object that is returned by the immutable version of *m* in class *MyT$Immutable*. So, in this case, *op*(t_1,…,t_n) is translated into an expression of the form *MyT$Immutable.m(…)*.

A more complex translation is required in cases where method *m* is not *void*. Let us consider, for instance, the translation of the term *size(s)* from the specification *StackSpec* in Figure 12a). According to the refinement mapping in Figure 16, the operation *size(s:Stack):Integer* is implemented by the method *int size()* of *StringStack*. The fact that *int* is the type chosen for implementing the sort *Integer*, is consistent with the assumption that the value of term *size(s)* is the value returned by the invocation of the method *size* and, hence, the term is translated to *StringStack$Immutable.size(s).value()* (recall from Section 3 that the immutable version of any non-*void* method returns a <*value, state*> pair).

The translation of term *top(t)* is similar. In this case, the operation *top(s:Stack):Elem* is implemented by the method *String top()* of *StringStack* and because *String* is the type chosen to implement sort *Elem*, the value of term *top(s)* is the value returned by the invocation of the method *top* and, hence, the term is translated to *StringStack$Immutable.top(s).value()*.

The translation of term *pop(t)* illustrates a yet different case. The operation *pop(s:Stack):Stack* is implemented by method *String pop()* of *StringStack*. The fact that the return type of this method is neither *void* nor *StringStack* (the type that was chosen to implement sort *Stack*), means that the stack denoted by *pop(t)* is captured by the state component that results from the invocation of method *pop* and, hence, the term will be translated into an expression of the form *StringStack$Immutable.pop(…).state()*. In this case, the *value()* component of the pair returned by *StringStack$Immutable.pop(…).state()* invocation, is of no interest in what contracts are concerned because there are no axioms that cover the meaning the implementer gives to that value.

Another point of complexity in the translation of axioms into contracts is the translation of strong equality used in axioms. As mentioned in Section 4, operation symbols can be interpreted by partial functions and, hence, terms may be undefined. 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. As such, equality testing within contracts must be consistent with this definition, that is, within pre and post-conditions, the evaluation of *equals*(t_1,t_2) (for testing value equality) should only be performed if it is not the case that t_1 and t_2 are both undefined. If t_1 and t_2 are both undefined then the equality $t_1=t_2$ is considered to hold.

Due to the fact that predicates in a specification cannot be partial, and that contracts of methods invoked within contracts are not monitored by JML runtime assertion checker, we have to avoid invoking, in our contracts, methods that implement ADT predicates in cases where their arguments are undefined.

A *def* function is defined and used in the translation process that supplies the definedness conditions for both terms and formulae of our specification language. 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* (if *op* is partial).

The translation of terms and formulae that are relevant in our setting is formally defined below. In order to simplify the presentation, all translation rules are defined from the perspective of a specific specification $\mu(\nu)$ in module μ. The translation of every term or formula is accomplished within the context of a given axiom – an axiom of specification $\mu(\nu)$.

Translation of Terms *(extended with constructs (C)x, \result, \result.value(), and \result.state())*: The translation – $[\![t]\!]$ – of a term *t* is defined by induction in the structure of *t*:

$$\begin{aligned}
[\![x]\!] &= x; && \\
[\![op(t_1,\ldots,t_n)]\!] &= C\$Immutable.\ m(\ [\![t_1]\!],\ldots,\ [\![t_n]\!]) && \text{if } C_r \text{ is } void \\
[\![op(t_1,\ldots,t_n)]\!] &= C\$Immutable.\ m(\ [\![t_1]\!],\ldots,\ [\![t_n]\!]).value() && \text{if } C_r \text{ is } C_s \\
[\![op(t_1,\ldots,t_n)]\!] &= C\$Immutable.\ m(\ [\![t_1]\!],\ldots,\ [\![t_n]\!]).state() && \text{otherwise}
\end{aligned}$$

if $op:s_1,\ldots,s_n \Rightarrow s$, and $\rho(s^u) = <C_s,\beta_s>$, and $\rho(op^u) = <C,\beta>$, and $\beta(op)$ is method $C_r\ m(\ldots)$;

$$[\![op(t_1,\ldots,t_n)]\!] = exp(\ [\![t_1]\!]\ /x_1,\ \ldots\ [\![t_n]\!]\ /x_n)$$

if $\rho(op^u) = <C,\beta>$, and $\beta(op)$ is a Java expression $exp(x_1,\ldots,x_n)$;

$$\begin{aligned}
[\![(C)\ x]\!] &= (C)\ x; \\
[\![\backslash result]\!] &= \backslash result \\
[\![\backslash result.value()]\!] &= \backslash result.value(); \\
[\![\backslash result.state()]\!] &= \backslash result.state();
\end{aligned}$$

Definedness Condition: The formula, *def(t),* that defines the condition under which a term *t* can be evaluated depends on the structure of *t*:

$$\begin{aligned}
def(x) &= true; \\
def((C)\ x) &= true; \\
def(\backslash result) &= true; \\
def(\backslash result.value()) &= true; \\
def(\backslash result.state()) &= true; \\
def(op(t_1,\ldots,t_n)) &= def(t_1) \wedge \ldots \wedge def(t_n) \wedge\ \phi\,[t_1/x_1,\ldots,t_n/x_n]
\end{aligned}$$

where $op(x_1,\ldots,x_n)$ *if* ϕ is the (only) domain condition in specification $\mu(op^u)$ for *op*. When there is no domain condition for *op*, $\phi\,[t_1/x_1,\ldots,t_n/x_n]$ is *true*. We chose not to include $def(\phi\,[t_1/x_1,\ldots,t_n/x_n])$ in the definedness condition for operation calls due to the remotely possible existence of recursive domain conditions.
The formula, $def(\phi)$, that defines the condition under which a formula ϕ can be evaluated depends on the structure of ϕ:

$$\begin{aligned}
def(true) &= true;\\
def(\neg\phi) &= def(\phi);\\
def(\phi_1\ BinOp\ \phi_2) &= def(\phi_1) \wedge def(\phi_2);\\
def(pred(t_1,\ldots,t_n)\,) &= def(t_1) \wedge \ldots \wedge def(t_n)\\
def(t_1{=}t_n)\,) &= true
\end{aligned}$$

where *BinOp* stands for any of the binary operators $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$.

According to the above considerations, we adopted the following approach: in any contract where testing equality is required, we first check whether both terms are defined and, if this is the case, then invoke method *equals*.

Translation of Formulas: The translation, $[\![\phi]\!]$, of a formula ϕ is defined by induction in the structure of ϕ:

$$\begin{aligned}
[\![true]\!] &= true;\\
[\![\neg\phi]\!] &= !\ [\![\phi]\!];\\
[\![\phi_1\wedge\phi_2]\!] &= [\![\phi_1]\!]\ \&\&\ [\![\phi_2]\!];\\
[\![\phi_1\vee\phi_2]\!] &= [\![\phi_1]\!]\ ||\ [\![\phi_2]\!];\\
[\![\phi_1\Rightarrow\phi_2]\!] &= [\![\phi_1]\!]\ ==>\ [\![\phi_2]\!];\\
[\![\phi_1\Leftrightarrow\phi_2]\!] &= [\![\phi_1]\!]\ ==\ [\![\phi_2]\!];\\
[\![pred(t_1,\ldots,t_n)]\!] &= C\$Immutable.\ m(\ [\![t_1]\!], \ldots,\ [\![t_n]\!]\,).value()
\end{aligned}$$

if $\rho(pred^u) = < C,\beta >$, and $\beta(pred)$ is method *boolean m(…)*;

$$\begin{aligned}
[\![t_1 = t_2]\!] = {} & (!\ [\![def(t_1)]\!]\ \&\&\ !\ [\![def(t_2)]\!]\,)\ ||\\
& (\ [\![def(t_1)]\!]\ \&\&\ [\![def(t_2)]\!]\ \&\&\ C\$Immutable.equals(\ [\![t_1]\!],\ [\![t_2]\!]\,))
\end{aligned}$$

if ν is the name of the only specification where the sort of t_1 (and t_2) is defined, and $\rho(\nu) = < C,\beta >$, and C is a class;

$$\begin{aligned}
[\![t_1 = t_2]\!] = {} & !\,(\ [\![def(t_1)]\!]\ ||\ [\![def(t_2)]\!]\,)\ ||\\
& (\ [\![def(t_1)]\!]\ \&\&\ [\![def(t_2)]\!]\ \&\&\ [\![t_1]\!] ==\ [\![t_2]\!]\,)
\end{aligned}$$

if ν is the name of the only specification where the sort of t_1 (and t_2) is defined, and $\rho(\nu) = < C,\beta >$, and C is a primitive type.

Translation of *Domain Restrictions*:

$$[\![op(\vec{x}\,)\ if\ \phi]\!] = requires\ \ [\![\ def(\phi) \Rightarrow \phi]\!]$$

to be placed in the contract of method *m*, where $\rho(op^{u}) = < C,\beta >$, and $\beta(op)$ is method $C_r\ m(\vec{c}\ \vec{x}),$ and *C* is a class; if *C* is a primitive type, then the translation is not accomplished (it is not possible to attach a pre-condition to an operation of a primitive type).

Example: Domain restriction *pop(s): if ¬isEmpty(s)* in specification *StackSpec* in Figure 12a) produces the pre-condition

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

in method *String$Pair pop (StringStack$Original s)* of class *StringStack$Immutable.*

Translation of axioms about *Constructors* and *Observers*:

$$
\begin{aligned}
[\![\, \phi \Rightarrow opC'(opC(\vec{x}), \vec{t}\,)=t]\!] &= ensures\ \ [\![\ def(\phi) \wedge \phi \Rightarrow opC'(r, \vec{t}\,)=t]\!]\,; \\
[\![\, \phi \Rightarrow opO(opC(\vec{x}), \vec{t}\,)=t]\!] &= ensures\ \ [\![\ def(\phi) \wedge \phi \Rightarrow opO(r, \vec{t}\,)=t]\!]\,; \\
\\
[\![\, \phi \Rightarrow predO(opC(\vec{x}), \vec{t}\,)]\!] &= ensures\ \ [\![\, def(\phi) \wedge\ \phi \wedge \\
&\qquad\qquad (def(predO(r, \vec{t}\,)) \Rightarrow\ predO(r, \vec{t}\,))]\!]\,; \\
[\![\, \phi \Rightarrow \neg predO(opC(\vec{x}), \vec{t}\,)]\!] &= ensures\ \ [\![\, (def(\phi) \wedge\ \phi \wedge \\
&\qquad\qquad def(\neg predO(r, \vec{t}\,))) \Rightarrow \neg predO(r, \vec{t}\,)]\!]\,;
\end{aligned}
$$

to be placed in the contract of method *m*, where $\rho(opC^{u})= < C,\beta >$, and $\beta(opC)$ is method $C_r\ m(\vec{c}\ \vec{x}),$ and *C* is a class, and *r* is i) *\result* if C_r is *void*; ii) *\result.value()* if C_r is *C*; and iii) *\result.state()* otherwise; if *C* is a primitive type, then the translation is not accomplished (it is not possible to attach a post-condition to an operation of a primitive type).

Example: Let us consider the axioms about observers included in the specification *StackSpec* in Figure 12a). By applying the translation rules above, and simplifying boolean expressions, we obtain the following contracts for the static methods of class *StringStackImmutable*:

– Axiom *top(push(_,i)) = i* produces the post-condition

```
    ensures !isEmpty(\result).value())&&
            String$Immutable.equals(top(\result).value(), i);
```

  in method *StringStack$Original push (StringStack$Original s, String i).*

– Axiom *pop(push(s,_)) = s* produces the post-condition

```
    ensures !isEmpty(\result).value())&&
            equals(pop(\result).state(), s).value();
```

  in method *StringStack$Original push (StringStack$Original s, String i).*

– Axiom *size(clear(_)) = zero(_)* produces the post-condition

```
    ensures size(\result).value() == 0;
```

  in method *StringStack$Original clear (StringStack$Original s).*

– Axiom *size(push(s,i)) = suc(size(s))* produces the post-condition

```
    ensures size(\result).value() == size(s).value() + 1;
```

  in method *StringStack$Original push (StringStack$Original s, String i).*

For simplicity, we omitted the *StringStack$Immutable* target in all calls to this class's methods and simplified all boolean expressions (the ones that result from the rigorous application of the formulas are fully presented in Appendix A).

Translation of axioms about *Derived* operations and predicates:

$$[\![\phi \Rightarrow opD(\vec{x})=t]\!] = ensures\ [\![def(\phi) \wedge \phi \Rightarrow \backslash result = t]\!]; \quad \text{if } C_r \text{ is } void$$
$$[\![\phi \Rightarrow opD(\vec{x})=t]\!] = ensures\ [\![def(\phi) \wedge \phi \Rightarrow \backslash result.value() = t]\!]; \quad \text{if } C_r \text{ is } C_s$$
$$[\![\phi \Rightarrow opD(\vec{x})=t]\!] = ensures\ [\![def(\phi) \wedge \phi \Rightarrow \backslash result.state() = t]\!]; \quad \text{otherwise}$$

to be placed in the contract of method m, if $opD{:}s_1,\ldots,s_n \Rightarrow s$, and $\rho(s^u)= <C_s,\beta_s>$, and $\rho(opD^u)= <C,\beta>$, and $\beta(opD)$ is method $C_r\ m(\ldots)$, and C is a class;

$$[\![\phi \Rightarrow predD(\vec{x})]\!] = ensures\ [\![def(\phi) \wedge \phi \Rightarrow \backslash result.value()]\!];$$
$$[\![\phi \Rightarrow \neg predD(\vec{x})]\!] = ensures\ [\![def(\phi) \wedge \phi \Rightarrow \neg \backslash result.value()]\!];$$

to be placed in the contract of method m, if $\rho(predD^u) = <C,\beta>$, and $\beta(predD)$ is method $boolean\ m(\vec{c}\ \vec{x})$, and C is a class; if C is a primitive type, then the translation is not accomplished (it is not possible to attach a post-condition to an operation of a primitive type).

Example: Let us now consider the axioms of *StackSpec* about derived operations and predicates. By applying the rules above we obtain the following contracts for the static methods of class *StringStack$Immutable*:

Axioms

$isEmpty(s) \Leftarrow size(s)= zero(\_)$

$\neg isEmpty(s) \Leftarrow \neg(size(s)= zero(\_))$

translate, respectively, to the post-conditions

```
ensures size(s).value() == 0  ==> \result.value();
ensures !(size(s).value() == 0) ==> !\result.value();
```

to be placed in the method *boolean$Pair isEmpty (StringStack$Original s)*.

Translation of axioms about equality:

Axioms of the form $(\phi \Rightarrow x_1=x_2)$ that allow us to express equivalence classes within the ADT are translated into contracts for the *equals* method.

$$[\![\phi \Rightarrow x_1=x_2]\!] = ensures\ [\![def(\phi) \wedge \phi \Rightarrow \backslash result]\!];$$

to be placed in the contract of method *boolean equals(C$Original x_1, Object x_2)* in class *C$Immutable*, where C is the class that implements the ADT where the axiom is defined.

In general, the contracts generated by the rules presented so far, are not final. The last step of the translation consists in the closure, through universal quantification of every free variable.

Closure of 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 (a suitable range is adopted).

5.2 Enforcing Equality Properties of Equational Logic

The contracts generated by our tool make use of cloning and equality and, hence, our methodology for checking whether an implementation behaves in conformance with a specification strongly relies on the execution of the *clone* method. It is essential that programmers ensure correct implementation of this method. Although our tool generates contracts for these methods, the soundness of the approach can be compromised if the implementations of these methods do not meet the following correctness criteria:

- *clone* method is required not to have any effect whatsoever on *this*;
- the implementation of *clone* is required to go deep enough in the structure of the object so that any shared reference with the cloned object cannot get modified through the invocation of any of the methods that implement the ADT operations. 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 as well as the array itself.

The contracts generated for *equals* and *clone* methods are as follows. Post-conditions are automatically generated for the *boolean$Pair equals(C$Original one, Object other)* method in class *C$Immutable* (where *C* is the class that implements the ADT where the axiom is defined), that test the results given by every observer operation and predicate when applied to two objects considered equal. This amounts to suppose the equational theories:

$$(x_1 = x_2) \Rightarrow (\, op(\vec{x}_1) = op(\vec{x}_2)\,)$$

$$(x_1 = x_2) \Rightarrow (\, pred(\vec{x}_1) \Leftrightarrow pred(\vec{x}_2)\,)$$

for each and every observer operation *op* and predicate *pred.*

A contract for the *Object clone(C$Original o)* method, also automatically generated, expresses that the cloned object must equal the original one (this equality is tested using method *equals*).

Contracts for *equals* and *clone*

For every observer operation and predicate $opp{:}s_1,\ldots,s_n \Rightarrow s$,

ensures \result ==> other instanceof C$Original &&
$$[\![\, (opp(one, \vec{x}\,) \;=\; opp((C\$Original)\; other, \vec{x}\,))\,]\!]$$

is placed in the contract of method *boolean$Pair equals(C$Original one, Object other)* in class *C$Immutable*, where *C* is the class that implements the ADT where the axiom is defined, and the post-condition

ensures equals(\result,o)

in the contract of method *Object clone(C$Original o)* in class *C$Immutable*, where *C* is the class that implements the given ADT.

Example: In our example, the following contracts are generated for the *equals* and *clone* methods of the class *StringStack$Immutable*:

```
/*@ ensures \result.value() ==> other instanceof StringStack$Original &&
     (!!isEmpty(one).value() &&
                               !!isEmpty((StringStack$Original) other).value())
     ||
     (!isEmpty(one).value() && !isEmpty((StringStack$Original) other).value()
     && String$Immutable.equals(top(one).value(),
                               top((StringStack$Original) other).value()));
  @ ensures \result.value() ==> other instanceof StringStack$Original &&
     (!!isEmpty(one).value() &&
                               !!isEmpty((StringStack$Original) other).value())
     ||
     (!isEmpty(one).value() && !isEmpty((StringStack$Original) other).value()
     && equals(pop(one).state(),
                        pop((StringStack$Original) other).state()).value());
  @ ensures \result.value() ==> other instanceof StringStack$Original &&
            size(one).value() == size((StringStack$Original) other).value());
   @*/
static public /*@ pure @*/ boolean equals(StringStack$Original one, Object
other)


/*@
  @ ensures equals(\result, other).value();
  @*/
static public Object clone(StringStack$Original other)
```

6 Benchmarking

We have tested our architecture on three data types, namely,

- The stack ADT described in this paper, with *Stack* refined into an array-based "standard" class, and *Element* refined into *java.lang.String*.
- The stack ADT described in this paper, with *Stack* refined into *java.util.Stack*, and *Element* refined into *java.lang.Object*.
- A data type representing rational numbers, with *Rational* refined into an immutable class represented by a pair of integers.

The source code for the three test cases are presented in the Appendices. They are also available in the url [14]. For each data type a few classes where used: the user's class (say, *IntRational$Original.java*, or *java.util.Stack.java*), the class responsible for checking the contracts (say, *IntRational$Immutable.java*), the class that replaces the user's class in our architecture (say, *IntRational.java*), and the various required *Pair* classes (say, *Object$Pair.java*, *int$pair.java*, or *boolean$Pair.java*).

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. Each data type was subjected to 1.000.000 randomly chosen operations, issued from a further class (say, *RationalRandomTest.java*). For each data type we assessed the time and space used in four different cases:

1. The user's class only, 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 Sun's Java compiler. We aim at understanding the overhead imposed by using the JML compiler on the architecture without contracts.
4. As above but with all contracts in place. This is how a user would experience our tool.

The results, average of ten runs, are as follows.

	1st case		*2nd case*		*3rd case*		*4th case*		*Total slowdown*
Package	*sec*	*KB*	*sec*	*KB*	*sec*	*KB*	*sec*	*KB*	
StringStack	2.71	600	3.26	600	11.58	604	21.21	604	7.8
java.util.Stack	2.27	600	4.35	603	10.66	606	23.07	605	10.2
IntRational	2.97	597	4.7	597	26.76	601	38.06	602	12.8

Inspecting the numbers for the first and the fourth cases one concludes that our architecture imposes no further space requirements, and that monitoring introduces a 10-fold time penalty, which we find plainly justifiable. 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 roughly half of the total overhead reported in the last column is due to contract monitoring alone, while the other half to the fact that we are using the JML compiler.

It should also be remarked that the tests were conducted with the contracts as generated by the rules in Section 5. A brief inspection of the contracts in, say, class *IntRational$Immutable.java*, Appendix C, reveals lots of room for boolean expression optimizations.

Finally a word on monitoring open assertions, that is assertions that contain variables not included in the parameters of the method. As described in Section 5.2, we use a JML *\forall* assertion. In this case, and since no range is generated for the contracts, no runtime monitoring is accomplished for these assertions.

We have also conducted preliminary tests in monitoring *\forall* assertions, by collecting all objects that enter or leave our tool (that is all objects passed as parameters or returned from the methods of the client's class), and using them as the required range. Setting up a limit of 100 elements in the collection (hence in the range of the *\forall*), we have experienced a total 50-fold slowdown.

7 Related Work

In this section, we examine the related work on run-time validation of implementations against their specifications.

In [13] a tool is presented that allows checking the behavioural 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 the language of [13] is tailored to the specification of properties of OO implementations 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. In order to illustrate this, we present below the specification of two properties of linked lists as they are presented in [13] as they would be specified in our approach.

```
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) ⇐ i>zero(_)
```

The first property states that *removeLast* operation provides the last element that was added to the list. The second property defines 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 the language of [13] 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 operations. Our specification language allows, within the specification of an ADT *T*, the expression of the application of operations to instances of any of *T*'s imported sorts; thus, whenever those operations are to be implemented as procedures, the state of those objects – which will most certainly be implemented as instance variables of the class that implements *T* – will eventually change. The tool generates contracts that allow the monitoring of those operations executions.

Antoy and Hamlet present [1] a testing approach for modules using an algebraic specification as a set of executable rewrite rules. The user supplies the specification, an implementation class, and an explicit mapping from concrete data structures of the implementation instance variables 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 direct implementation is composed of code that is generated from the specification by representing instances of abstract data types as terms, and manipulating them according to the rewrite rules defined in the ADT specification.

The representation mapping, or abstraction mapping, must be written by the user in the same language as the implementation class, and asks user knowledge about internal representation details. Here lies a difference between this and our approach: 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 this 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. In what concerns our approach, these kind of axioms are acceptable: they originate post-conditions in the method that implements the *insert* operation.

We further believe that the rich structure that our specifications can present, together with the possibility to, through refinement mappings, map a same module into many different packages all of which implement the same specification, is a positive point in our approach that we cannot devise in the above referred approaches.

The Daistish tool [16] is a PERL script that processes an algebraic, functional, specification of an ADT, along with the code for an object implementing the ADT, to produce a test driver. Appropriate data points (values for the parameters to the operations that are called in the axioms) are selected, values for both sides of the axioms are separately computed, and results are compared. Test vectors given by the user are used to define the parameter values. An implementation testing succeeds when equivalent values are produced for each side of the axioms. Versions exist for Eiffel and C++.

The fundamental difference between this approach and ours is that their specifications must already contain some information concerning implementation: the types of arguments and result in function signatures must reveal whether a mutable or immutable implementation is expected. Furthermore, testing values for elements of the several sorts must be supplied by the user in order to the Daistish tool to work properly. In what concerns equality, the paper does not clarify which semantics is used; it is only said that "if an axiom is written using the "=" symbol, Daistish will generate code employing the algebraic operation appropriate for the types being compared".

The belief behind the MOP (Monitoring-oriented programming) formal framework for software development and analysis [5], is that specification and implementation should together form a system and interact with each other. This framework is independent from any particular specification formalism. These can be modularly added to the MOP framework, provided that they are coded as logic plug-ins, that is, modules whose interfaces respect some standardized conventions, and that incorporate a monitor synthesis algorithm. This algorithm takes formal specifications and produces corresponding concrete monitoring code that analyzes program execution traces. Runtime violations and validations of specifications may result in adding functionality to the system by executing user-defined code at user-defined places in the program: executing recovery code, outputting or sending messages, throwing exceptions, etc.

A WWW repository exists with some downloadable logic plug-ins – future time and past-time temporal logics, extended regular expressions and JASS – and support for JML annotations is intended for soon. Due to the fact that our tool generates JML annotated classes, we may envisage these classes as input data to the JAVA-MOP framework with JML support.

Further work exists on runtime checking of specifications against implementations adopting a model based approach to specification. As stressed in Section 2, the representation mapping, or abstraction mapping, that the implementer must supply turns the task of the implementer more difficult. In our approach, the contract generation process does not need to know any details of state implementation whatsoever, because it only works with method calls. Then, this kind of abstraction function/mapping is not needed, thus lightening the burden of the implementer. The frameworks we describe below are among those model based ones.

Barnett and Schulte present a method for specifying interfaces uses the language ASML to write an executable specification – ASML specifications are model programs, that is, they are operational specifications of the behaviour expected of any implementation [2]. This specification defines the behaviour of a component, as seen through its interface, by a client. The component implementing the interface, and its specification, are run concurrently with no need for any sort of instrumentation at all, and in a way that is transparent to the client; this is accomplished through the use of a component which operates as a proxy, and that forks all the calls from the client to the implementation component, so that they are also delivered to the specification component. If all pairs of results agree, then, for that particular trace, the component is a behavioural subtype of the specification.

Edwards et.al.'s general strategy for automated black-box testing of software components is presented that combines three techniques: automatic generation of component test-drivers, automatic generation of test data, and automatic or semi-automatic generation of wrappers that have the same interface as the base component [6,7]. The approach to the specification of component interfaces is model-based – the language *Resolve* was chosen – but semi-formal or informal behavioural descriptions are also accepted (informal descriptions require more human interaction, however). Here again the built-in test (BIT) wrappers are transparent to client and component code, and can be inserted and removed without changing client code. Pre and post-conditions and abstract invariant checks are written in terms of the component's state abstract mathematical model. Difficulties arise when generating code for checking assertions containing quantifiers: it cannot be fully automated. Human intervention is one of the three described possible solutions for these cases.

The SLAM system [12] allows the user to specify a program in a high level specification language – an object oriented formal specification language that integrates algebraic specifications with model-based ones. Executable and readable code written in an object-oriented programming language can be generated from the program formal specification. The code contains runtime checkable assertions corresponding to some of the pre and post-conditions of the specification (the ones that were declared as *checkable*). These assertions are complex logical formulae and Prolog programs that check them are automatically generated. A function specification is a pair pre and post-condition that indicates the relationship between the result and the arguments. Whenever possible, the specifier may also define a computable expression that defines the result.

SLAM unifies algebraic and model-based languages by specifying operations through logical pre and post-conditions, but restricting logical formulae to a computable view of quantifiers as traversal operations on data.

8 Conclusions and Further Work

We described an approach for testing Java implementations of abstract data types against their specifications. We adopted an algebraic, property-driven, approach to ADT specifications rather than a model-driven one. Although we recognise the important role played by model-based approaches to ADT specification, we also think that, for a significant part of software developers, it can be rather difficult to write model-based program specifications – the representation mapping, or abstraction mapping, that the implementer must supply is far from trivial. We believe that the simplicity of property-driven specifications can encourage more software developers to use formal specifications. Therefore, we find it important to equip property-driven approaches with tools similar to the ones currently available for model-driven approaches.

Specific features of the language we adopted for specifying abstract data types, such as the classification of operations in specific categories, and strong restrictions in the form of the axioms, not only simplify the task of creating specifications, but are also effective in driving the automatic identification of contracts for implementing classes.

Abstract data types are specified through modules that gather the component specifications that are needed to completely specify all the needed sorts and operations. A module provides names for its component specifications, which are then mapped into Java types in the context of refinement mappings. These are easy to build insofar as essentially only signatures – of adt operations and of Java methods or primitive operators – are required to describe the relation between ADT specification components and operations and corresponding Java types and methods.

The notion of module lets us represent from the most simple, basic ADT, such as the closed specification in Figure 9, to very rich and complex structures. Moreover, we are able to map a same module, through different refinement mappings into many different Java packages which all implement the same specification.

Further as well as ongoing work covers extension to the methodology and to the framework in order to cope with parametric specifications, specifications as an extension of others, and specifications with more than one intrinsic sort.

We adopted a semantics of strong equality for the equality symbol used in axioms, that is, either both sides are defined and are equal, or both sides are undefined. In addition to the contracts that are generated from user defined axioms, our tool also automatically generates contracts that are consistent with the adopted notion of equality.

Closure of assertions within the context of a pair assertion/method, as briefly described in section 5, implies the insertion of the *\forall* JML construct that quantifies over all variables that are free within the given context. A suitable range must be defined for every quantified variable, that is, a set of values that is representative of the type of the variable, as specified in the ADT. Ongoing work focuses on the automatic generation of this set of values by collecting all objects that enter or leave our tool (that is all objects passed as parameters or returned from the methods of the client's class), and using them as the required range.

Due to its syntax directed nature, contract generation produces conditions that have a number of redundant parts – *true && true*, for example. We aim at the simplification of generated contracts through the simplification of Boolean expressions.

Acknowledgements

Thanks are due to José Luiz Fiadeiro for many fruitful discussions that have helped to set this work up which was partially supported through the POSI/CHS/48015/2002 Project *Contract Guided System Development.*

References

1. S.Antoy and D.Hamlet, "Automatically Checking an Implementation against its Formal Specification", *Software Engineering*, 26(1), 55-69, 2000.
2. M.Barnett and W.Schulte, "Spying Components: A Runtime Verification Technique", *Proc. Workshop on Specification and Verification of Component Based Systems* - OOPSLA 2001, Tampa, Florida, USA, 2001. ACM Press.
3. D.Bartetzko, C.Fischer, M.Möller, and H.Wehrheim, "Jass – Java with Assertions", *Electronic Notes in Theoretical Compuer Science* **55** (2), July 2001.
4. M.Blum and H.Wasserman, "Software Reliability via run-time result-checking", *Journal of the ACM* **44** (6), 826-849, November 1997.
5. F.Chen, M. d'Amorim, and G. Rosu, "A Formal Monitoring-based Framework for Software Development and Analysis", *Proc. 6th International Conference on Formal Engineering Methods* (ICFEM'04), 357-372, 2004.
6. S.H.Edwards, "A framework for practical, automated black-box testing of component-based software", *Software Testing, Verification and Reliability*, **11**(2), 97-111, June, 2001.
7. S.H.Edwards, G.Shakir, M. Sitaraman, B.W.Weide and J. Hollingsworth, "A framework for detecting interface violations in component-based software", *Proceedings of the International Conference on Software Reuse*, 46–55, IEEE Computer Society Press, 1998.
8. M. Bidoit and P.D. Mosses, *CASL User Manual*, (eds), LNCS 2900, Springer, 2004
9. CoFI (The Common Framework Initiative), *Casl Reference Manual*, P.D.Mosses (ed), LNCS 2960, Springer, 2004.
10. J.Goguen, J.W.Thatcher, and E.Wagner, "An initial algebra approach to the specification, correctness, and implementation of abstract data types", in R. T. Yeh, editor,

Current Trends in Programming Methodology, IV, Data Structuring, 80-149, Prentice-Hall, 1978. also IBM Research Report RC 6487 (1976).

11. J.V.Guttag, J.J.Horning, S.J.Garland, K.D.Jones, A.Modet, and J.M.Wing, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
12. A.Herranz-Nieva and J.Moreno-Navarro, “Generation of and Debugging with Logical pre and pos Conditions”, *Proceedings of the Fourth International Workshop* on *Automated and Algorithmic Debugging*, Munich, August 2000.
13. J.Henkel and A.Diwan, “A Tool for Writing and Debugging Algebraic Specifications”, *Proceedings of the International Conference on Software Engineering (ICSE) 2004*, Scotland 2004.
14. http://labmol.di.fc.ul.pt/congu/
15. http://www.di.fc.ul.pt/~vv/projects/contracts/
16. M.Hughes and D.Stotts, “Daistish: Systematic Algebraic Testing for OO Programs in the presence of side-effects”, *Proceedings of The International Symposium on Software Testing and Verification*, 53-61, ACM, 1996.
17. M.Karaorman, U.Holzle and J.Bruno, ”jContractor: A reflective Java library to support design by contract”, *Proceedings of Meta-Level Architectures and Reflection*, LNCS 1616, Springer-Verlag, 1999.
18. R.Kramer, “iContract - The Java Design by Contract Tool”, *Proceedings of TOOLS USA* '98 conference, IEEE Computer Society Press, 1999.
19. G.T.Leavens, K.Rustan M.Leino, Erik Poll, Clyde Ruby, and Bart Jacobs, “JML: notations and tools supporting detailed design in Java”, *OOPSLA'00 Companion*, Minneapolis, Minnesota, 105-106, ACM Press, 2000.
20. G.T.Leavens, A.L.Baker, and C.Ruby. “Preliminary design of JML: A behavioural interface specification language for Java”, Technical Report 98-06-rev27, Iowa State University, Department of Computer Science, April 2005.
21. B.Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice-Hall PTR, ISBN 0-13-629155-4, 1997.
22. The Java Modeling Language examples page: http://www.cs.iastate.edu/~leavens/JML/-examples.shtml
23. T.Skotiniotis and D.Lorenz. “Cona: aspects for contracts and contracts for aspects”, *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 196-197. ACM Press, 2004.
24. J.M.Spivey, *The Z Notation: A Reference Manual*, International Series in Computer Science, Prentice-Hall, 1992.

Appendix A

In this section we put together the complete example introduced in the main text and used throughout section 5. We take the module for a stack (Figure 13) and an implementation of a stack of *String* objects centred on the class *StringStack* (Figure 15). We propose a mutable implementation – notice the *void* return type of the constructors, and the mutable implementation of the *pop* method that returns the top element. The refinement mapping (Figure 16) states that the elements of the stack are *java.lang.String* objects whether the primitive type *int* is chosen to describe the size of the stack.

Figure 17 below shows the immutable class *StringStack$Immutable* that is automatically created by our tool. It has a static method for each method in class *StringStack*. Contracts are added to the corresponding methods according to the rules defined in the previous sections.

The class *String$Immutable* is also automatically generated; due to the fact that the closed specification *ElemSpec* in Figure 12(b) does not define any axioms whatsoever, this immutable class has no contracts.

```
/**
 *
 */
public class StringStack$Immutable {

   /*@ ensures !(true && true) && !(true && true) ||
                (true && true) && (true && true) && size(\result).value() == 0;
     @*/
   static public StringStack$Original clear(StringStack$Original s) {
         StringStack$Original result = (StringStack$Original) clone(s);
         result.clear();
         return result;
   }

   /*@ ensures !(true && !isEmpty(\result).value()) && !true ||
               (true && !isEmpty(\result).value()) && true &&
               String$Immutable.equals(top(\result).value(), i);

     @ ensures !(true && !isEmpty(\result).value()) && !true ||
               (true && !isEmpty(\result).value()) && true &&
               equals(pop(\result).state(), s).value();

     @ ensures !(true && true) && !(true && true && true) ||
               (true && true) && (true && true && true) &&
               size(\result).value() == size(s).value() + 1;
    */
   static public StringStack$Original push(StringStack$Original s,String i) {
         StringStack$Original result = (StringStack$Original) clone(s);
         result.push(i);
         return result;
   }

   /*@ requires true ==> !isEmpty(s).value();
     @*/
   static public /*@ pure @*/ String$Pair pop(StringStack$Original s) {
         StringStack$Original clone = (StringStack$Original) clone(s);
         String$Pair result = new String$Pair(clone.pop(), clone);
         return result;
   }
```

```
    /*@ requires true ==> !isEmpty(s).value();
      @*/
    static public /*@ pure @*/ String$Pair top(StringStack$Original s) {
          StringStack$Original clone = (StringStack$Original) clone(s);
          return new String$Pair(clone.top(), clone);
    }

    /*
     */
    static public /*@ pure @*/ int$Pair size(StringStack$Original s) {
          StringStack$Original clone = (StringStack$Original) clone(s);
          return new int$Pair(clone.size(), clone);
    }

    /*@ ensures true && (!(true && true) && !(true && true) ||
                  (true && true) && (true && true) &&
                  size(s).value() == 0)  ==> \result.value();
      @ ensures true && (!(!(true && true) && !(true && true) ||
                  (true && true) && (true && true) &&
                  size(s).value() == 0)) ==> !\result.value();
      @*/
    static public /*@ pure @*/ boolean$Pair isEmpty(StringStack$Original s) {
          StringStack$Original clone = (StringStack$Original) clone(s);
          return new boolean$Pair(clone.isEmpty(), clone);
    }

    /*@ ensures \result.value() ==> other instanceof StringStack$Original &&
                  (!(true && !isEmpty(one).value()) &&
                  !(true && !isEmpty((StringStack$Original) other).value()) ||
                  (true && !isEmpty(one).value() && true) &&
           (true && !isEmpty((StringStack$Original) other).value() && true) &&
           String$Immutable.equals(top(one).value(),
                           top((StringStack$Original) other).value()));
      @ ensures \result.value() ==> other instanceof StringStack$Original &&
                  (!(true && !isEmpty(one).value()) &&
                  !(true && !isEmpty((StringStack$Original) other).value()) ||
                  (true && !isEmpty(one).value()) &&
                  (true && !isEmpty((StringStack$Original) other).value()) &&
                  equals(pop(one).state(),
                          pop((StringStack$Original) other).state()).value());
     @ ensures \result.value() ==> other instanceof StringStack$Original &&
                  (!(true && true) && !(true && true) ||
                  (true && true) && (true && true) &&
            size(one).value() == size((StringStack$Original) other).value());
          @*/
          static public /*@ pure @*/ boolean$Pair equals(
                                        StringStack$Original one, Object other){
                  StringStack$Original clone = (StringStack$Original) clone(one);
                  return new boolean$Pair(clone.equals(other), clone);
          }

          /*@
            @ ensures equals(\result, other).value();
            @*/
          static public Object clone(StringStack$Original other) {
                  return other.clone();
          }

    }

public class String$Immutable {

    public /*@ pure @*/ static boolean equals(String one, String other) {
        return one == null? other == null : one.equals(other);
    }
}
```

Fig. 17. Immutable classes generated by the tool

Both immutable and wrapper classes need to use pairs of elements in order to cope with non-*void* methods, as already explained in section 3. These classes are automatically created by the tool and there is one for each different method return type. In this *StringStack* example three classes are created (Figure 18) that denote pairs in which the state element is a stack and the value element is a primitive integer, a string, and a boolean.

```
**
 *
 */
public class int$Pair {

        private int value;
        private StringStack$Original state;

        public int$Pair(int v, StringStack$Original s) {
                state = s;
                value = v;
        }

        public /*@ pure @*/ int value() {
                return value;
        }

        public StringStack$Original state() {
                return state;
        }
}

**
 *
 */
public class boolean$Pair {

        private boolean value;
        private StringStack$Original state;

        public boolean$Pair(boolean v, StringStack$Original s) {
                state = s;
                value = v;
        }

        public /*@ pure @*/ boolean value() {
                return value;
        }

        public StringStack$Original state() {
                return state;
        }
}

**
 *
 */
public class String$Pair {

        private String value;
        private StringStack$Original state;

        public String$Pair(String v, StringStack$Original s) {
                state = s;
                value = v;
        }
        public /*@ pure @*/ String value() {
                return value;
        }

        public /*@ pure @*/ StringStack$Original state() {
                return state;
        }
}
```

Fig. 18. Auxiliary classes used in immutable and wrapper classes

Figure 19 presents the wrapper class that our tool creates and that has the same name as the concrete implementation class provided by the user – in this case, *StringStack*. This wrapper class has all the methods of the original class. These methods invoke the corresponding contract equipped methods in class *StringStack$Immutable* class that, in turn, invoke the original ones. The wrapper class comprises a private *StringStack$Original* attribute, used as an argument on the invocation of every immutable corresponding method.

```
/**
 *
 */
public class StringStack implements Cloneable {

    protected StringStack$Original wrappedObject;

    public StringStack() {
        wrappedObject = new StringStack$Original();
    }

    public void clear() {
         wrappedObject = (StringStack$Original) StringStack$Immutable
                                 .clear(wrappedObject);
    }

    public void push(String item) {
         wrappedObject = (StringStack$Original)
                         StringStack$Immutable.push(wrappedObject, item);
    }

    public String pop() {
        String$Pair result = StringStack$Immutable.pop(wrappedObject);
        wrappedObject = result.state();
        return result.value();
    }

    public boolean isEmpty() {
        boolean$Pair result = StringStack$Immutable.isEmpty(wrappedObject);
        wrappedObject = result.state();
        return result.value();
    }

    public String top() {
        String$Pair result = StringStack$Immutable.top(wrappedObject);
        wrappedObject = result.state();
        return result.value();
    }

    public int size() {
        int$Pair result = StringStack$Immutable.size(wrappedObject);
        wrappedObject = result.state();
        return result.value();
    }
```

```
     public boolean equals(Object other) {
         boolean$Pair result = StringStack$Immutable.equals(
                             wrappedObject,(Object) unwrapCheck(other));
         wrappedObject = result.state();
         return result.value();
     }

    public Object clone() {
         return (Object) wrapCheck(
                             StringStack$Immutable.clone(wrappedObject));
    }

     // AUXILIARY METHODS FOR WRAPPING AND UNWRAPPING

     static private StringStack wrap(StringStack$Original obj) {
             StringStack result = new StringStack();
             result.wrappedObject = obj;
             return result;
     }

     static private Object wrapCheck(Object obj) {
             return (obj instanceof StringStack$Original)?
                             wrap((StringStack$Original) obj) : obj;
     }

     static private Object unwrapCheck(Object obj) {
             return (obj instanceof StringStack)?
                             ((StringStack) obj).wrappedObject : obj;
     }
}
```

Fig. 19. Wrapper class generated by the tool

Appendix B

This appendix contains the code for the *java.util.Stack* benchmarks used in Section 6. It contains the refinement mapping (Figure 20) of the *GenericStack* module specification (Figure 13) into classes *java.lang.Object* and *java.util.Stack*. It also contains two classes: the class that replaces the user's class in our architecture (Figure 21), and the classes responsible for checking the contracts (Figure 22). We have omitted the various *Pair* classes (*boolean$Pair.java, int$Pair.java, Object$Pair.java*).

```
refinement mapping
--------------------------------------------------------------------------
    IntegerSpec   is primitive int
         zero(x1:Integer)                   is 0
         suc(x1:Integer):Integer            is x1+1
         pred(x1:Integer):Integer           is x1-1
         _<_(x1:Integer, x2:Integer)        is x1<x2

    ElemSpec  is class java.lang.Object

    StackSpec is class java.util.Stack
         clear(s:Stack):Stack               is void clear()
         push(s:Stack,i:Elem):Stack         is void push(java.lang.Object i)
         pop(s:Stack):Stack                 is java.lang.Object pop()
         top(s:Stack):Integer               is java.lang.Object peek()
         size(s:Stack):Integer              is int size()
         isEmpty(s:Stack)                   is boolean empty()
```

Fig. 20. Refinement mapping

```
public class Stack extends java.util.Vector {

        protected java.util.Stack wrappedObject;

        public Stack() {
                wrappedObject = new java.util.Stack();
        }

        public void clear() {
                wrappedObject = Stack$Immutable.clear(wrappedObject);
        }

        public boolean empty() {
                boolean$Pair pair = Stack$Immutable.empty(wrappedObject);
                wrappedObject = pair.state();
                return pair.value();
        }

        public Object peek() {
                Object$Pair pair = Stack$Immutable.peek(wrappedObject);
                wrappedObject = pair.state();
                return (Object) wrapCheck(pair.value());
        }

        public Object pop() {
                Object$Pair pair = Stack$Immutable.pop(wrappedObject);
                wrappedObject = pair.state();
                return (Object) wrapCheck(pair.value());
        }

        public Object push(Object item) {
                Object$Pair pair = Stack$Immutable.push(
                                wrappedObject, (Object) unwrapCheck(item));
                wrappedObject = pair.state();
                return (Object) wrapCheck(pair.value());
        }

        public int size() {
                int$Pair pair = Stack$Immutable.size(wrappedObject);
                wrappedObject = pair.state();
                return pair.value();
        }

        public int search(Object o) {
                return wrappedObject.search((Object) unwrapCheck(o));
        }

        public boolean equals(Object other) {
                boolean$Pair pair = Stack$Immutable.equals(
                                wrappedObject, (Object) unwrapCheck(other));
                wrappedObject = pair.state();
                return pair.value();
        }

        public Object clone() {
                return (Object) wrapCheck(Stack$Immutable.clone(wrappedObject));
        }

        // AUXILIARY METHODS FOR WRAPPING AND UNWRAPPING

        static private Stack wrap(java.util.Stack obj) {
                Stack result = new Stack();
                result.wrappedObject = obj;
                return result;
        }

        static private Object wrapCheck(Object obj) {
                return (obj instanceof java.util.Stack)?
                                        wrap((java.util.Stack) obj) : obj;
        }

        static private Object unwrapCheck(Object obj) {
                return (obj instanceof Stack)? ((Stack) obj).wrappedObject : obj;
        }

        public String toString() {
                return wrappedObject.toString();
        }

        // All the methods of the type java.util.Stack that are not specified
        // on the ADT, inherited or not, should be replicated here

}
```

Fig. 21. Wrapper class

For each method implemented by *java.util.Stack* that does not correspond to an abstract data type operation, there should be a method in the wrapper class with the same signature. These methods, which we chose to omit here due to its number, should call the original method using the wrapped *java.util.Stack* object as target. This is necessary if one wants the wrapper class to be able to substitute the original class in what client classes are concerned.

```
public class Stack$Immutable {

      /*@ ensures !(true && true) && !(true && true) ||
              (true && true) && (true && true) && size(\result).value() == 0;
        @*/
    static public java.util.Stack clear(java.util.Stack s) {
        java.util.Stack result = (java.util.Stack) clone(s);
        result.clear();
        return result;
    }

    /*@ ensures !(true && !empty(\result.state()).value()) && !true ||
              (true && !empty(\result.state()).value()) && true &&
              Object$Immutable.equals(peek(\result.state()).value(), o);
        @ ensures !(true && !empty(\result.state()).value()) && !true ||
              !(true && empty(\result.state()).value()) && true &&
              equals(pop(\result.state()).state(), s).value();
        @ ensures !(true && true) && !(true && true && true) ||
              (true && true) && (true && true && true) &&
              size(\result.state()).value() == size(s).value() + 1;
      */
    static public Object$Pair push(java.util.Stack s, Object o) {
        java.util.Stack clone = (java.util.Stack) clone(s);
        Object$Pair result = new Object$Pair(clone.push(o), clone);
        return result;
    }

    /*@
      @ requires true ==> !empty(s).value();
      @*/
    static /*@ pure @*/ public Object$Pair pop(java.util.Stack s) {
        java.util.Stack clone = (java.util.Stack) clone(s);
        Object$Pair result = new Object$Pair(clone.pop(), clone);
        return result;
    }

    /*@
      @ requires true ==> !empty(s).value();
      @*/
    static /*@ pure @*/ public Object$Pair peek(java.util.Stack s) {
        java.util.Stack clone = (java.util.Stack) clone(s);
        return new Object$Pair(clone.peek(), clone);
    }

    static  /*@ pure @*/ public int$Pair size(java.util.Stack s) {
        java.util.Stack clone = (java.util.Stack) clone(s);
        return new int$Pair(clone.size(), clone);
    }

    /*@ ensures true && (!(true && true) && !(true && true) ||
                      (true && true) && (true && true) &&
                      size(s).value() == 0)  ==> \result.value();
      @ ensures true && (!(!(true && true) && !(true && true) ||
                      (true && true) && (true && true) &&
                      size(s).value() == 0)) ==> !\result.value();
      @*/
    static /*@ pure @*/ public boolean$Pair empty(java.util.Stack s) {
        java.util.Stack clone = (java.util.Stack) clone(s);
        return new boolean$Pair(clone.empty(), clone);
    }

    /*@ ensures \result.value() ==> other instanceof java.util.Stack &&
              (!(true && !empty(one).value()) &&
              !(true && !empty((java.util.Stack) other).value()) ||
              (true && !empty(one).value() && true) &&
              (true && !empty((java.util.Stack) other).value() && true) &&
               Object$Immutable.equals(peek(one).value(),
                                      peek((java.util.Stack) other).value()));
      @ ensures \result.value() ==> other instanceof java.util.Stack &&
               (!(true && !empty(one).value()) &&
               !(true && !empty((java.util.Stack) other).value()) ||
               (true && !empty(one).value()) &&
               (true && !empty((java.util.Stack) other).value()) &&
               equals(pop(one).state(),
                              pop((java.util.Stack) other).state()).value());
    @ ensures \result.value() ==> other instanceof java.util.Stack &&
               (!(true && true) && !(true && true) ||
               (true && true) && (true && true) &&
               size(one).value() == size((java.util.Stack) other).value());
    @*/
  static /*@ pure @*/ public boolean$Pair equals(
                                      java.util.Stack one, Object other) {
      java.util.Stack clone = (java.util.Stack) clone(one);
      return new boolean$Pair(clone.equals(other), clone);
  }

  /*@
    @ ensures equals(\result, o).value();
    @*/
  static public Object clone(java.util.Stack o) {
      return o.clone();
  }
}
```

Fig. 22. Immutable classes equipped with contracts

Appendix C

This appendix contains the code for the *Rational* benchmarks used in Section 6. It contains the specification for the *Rational* datatype (Figure 23), and the refinement mapping of this specification into class *IntRational.java* (Figure 24). It also contains two classes: the class that replaces the user’s class in our architecture (Figure 25), and the class responsible for checking the contracts (Figure 26). We have omitted the various *Pair* classes (*boolean$Pair.java, int$Pair.java, IntRational$Pair.java*).

```
specification

   sorts     Rational
   operations and predicates
        constructors
             make:   Rational int int -->? Rational;

        observers
             num: Rational --> int;
             den: Rational --> int;
        derived
             mult: Rational Rational --> Rational;
             div: Rational Rational -->? Rational;
             inverse: Rational -->? Rational;
             zero: Rational --> Rational;
        domains
             F, G: Rational; N, D: int;
             make (F, N, D)  if not (D = 0);
             inverse (F)     if not (num (F) = 0);
             div (F, G)      if not (num (G) = 0);
        axioms
             F, G, H: Rational; N, D: int;
             num(make(H, N, D)) = 0 if N = 0;
             den(make(H, N, D)) = num(make(H, N, D)) * D/N
                 if not(num(make(H, N,  D)) = 0) && not(D = 0) && not(N = 0);
             den(make(H, N, D)) = 1
                 if num(make(H, N,  D)) = 0 || D = 0 || N = 0;
             mult (F, G) = make (H, num (F) * num (G), den (F) * den (G));
             div (F, G)  = mult (F, inverse (G));
             inverse (F) = make (H, den (F), num (F));
             zero (F) = make (H, 0, 1);
             F = G  if  num(F) * den (G) = num (G) * den (F);
   end
```

Fig. 23. Specification of the Rational abstract data type

```
refinement mapping

Rational is class IntRational
  make(r: Rational, n:int,d:int):Rational   is IntRational make(int i1,int i2)
  num(r: Rational):int                      is int num( )
  den(r: Rational):int                      is int den( )
  mult(r1: Rational, r2: Rational):Rational is IntRational mult(IntRational r)
  div(r1: Rational, r2: Rational):Rational  is IntRational div(IntRational r)
  inverse(r:Rational):Rational              is IntRational inverse( )
  zero(r:Rational):Rational                 is IntRational zero( )
```

Fig. 24. Refinement mapping

```
/**
 */
  public class IntRational {

      protected IntRational$Original wrappedObject;

      public IntRational() {
          wrappedObject = new IntRational$Original();
      }

      public IntRational make(int num, int den) {
          IntRational$Pair pair =
                         IntRational$Immutable.make(wrappedObject, num, den);
          wrappedObject = pair.state();
          return (IntRational) wrapCheck(pair.value());
      }

      public int num() {
          int$Pair pair = IntRational$Immutable.num(wrappedObject);
          wrappedObject = pair.state();
          return pair.value();
      }

      public int den() {
          int$Pair pair = IntRational$Immutable.den(wrappedObject);
          wrappedObject = pair.state();
          return pair.value();
      }

      // requires other != null; - not needed; jml does it for you.
      public IntRational mult(IntRational other) {
          IntRational$Pair pair = IntRational$Immutable.mult(
                wrappedObject, (IntRational$Original) unwrapCheck(other));
          wrappedObject = pair.state();
          return (IntRational) wrapCheck(pair.value());
      }

      public IntRational inverse() {
          IntRational$Pair pair = IntRational$Immutable.inverse(wrappedObject);
          wrappedObject = (IntRational$Original) pair.state();
          return (IntRational) wrapCheck(pair.value());
      }

      // requires other != null; - not needed; jml does it for you.
      public IntRational div(IntRational other) {
          IntRational$Pair pair = IntRational$Immutable.div(
                wrappedObject, (IntRational$Original) unwrapCheck(other));
          wrappedObject = pair.state();
          return (IntRational) wrapCheck(pair.value());
      }

      public IntRational zero() {
          IntRational$Pair pair = IntRational$Immutable.zero(wrappedObject);
          wrappedObject = pair.state();
          return (IntRational) wrapCheck(pair.value());
      }

      public boolean equals(Object other) {
          boolean$Pair pair = IntRational$Immutable.equals(
                                 wrappedObject, (Object) unwrapCheck(other));
          wrappedObject = pair.state();
          return pair.value();
      }

      public Object clone() {
          return (Object) wrapCheck(IntRational$Immutable.clone(wrappedObject));
      }

      // AUXILIARY METHODS FOR WRAPPING AND UNWRAPPING

      static private IntRational wrap(IntRational$Original obj) {
          IntRational result = new IntRational();
          result.wrappedObject = obj;
          return result;
      }

      static private Object wrapCheck(Object obj) {
          return (obj instanceof IntRational$Original)?
              wrap((IntRational$Original) obj) : obj;
      }
```

```
        static private Object unwrapCheck(Object obj) {
            return (obj instanceof IntRational)?
                ((IntRational) obj).wrappedObject : obj;
        }
}
```

Fig. 25. Wrapper class

```
/**
 */
public class IntRational$Immutable {

    /*@
    @ requires true ==> !(!true && !true || true && true && den == 0);
    @ ensures true && (!true && !true || true && true && num == 0) ==>
             !(true && !(den == 0) && true && true && true) && !true ||
             (true && !(den == 0) && true && true && true) && true &&
             num(\result.value()).value() == 0;
    @ ensures (true && true && true) &&
            !(!(true && !(den == 0) && true && true && true) && !true ||
            (true && !(den == 0) && true && true && true) && true &&
            num(\result.value()).value() == 0) &&
             !(!true && !true || true && true && den == 0) &&
            !(!true && !true || true && true && num == 0)
            ==>
             !(true && !(den == 0) && true && true && true) &&
            !(true && (true && !(den==0) && true && true && true) && true && true)
            || (true && !(den == 0) && true && true && true) &&
            (true && (true && !(den == 0) && true && true && true) && true && true)
            && den(\result.value()).value()==num(\result.value()).value()*den/num;

   @ ensures (true && true && true) &&
             //###### num(make(H, N, D)) = 0 || D = 0 || N = 0
             ((!(true && !(den == 0) && true && true && true) && !true ||
             (true && !(den == 0) && true && true && true) && true &&
             num(\result.value()).value() == 0) ||
             (!true && !true || true && true && den == 0) ||
             (!true && !true || true && true && num == 0))
                 ==>
              !(true && !(den == 0) && true && true && true) && !true ||
             (true && !(den == 0) && true && true && true) && true &&
             den(\result.value()).value() == 1;
    @*/
     static public /*@ pure @*/ IntRational$Pair make(
                                IntRational$Original object, int num, int den) {
         IntRational$Original clone = (IntRational$Original) clone(object);
         return new IntRational$Pair(clone.make(num, den), clone);
     }

     static public /*@ pure @*/ int$Pair num(IntRational$Original object) {
         IntRational$Original clone = (IntRational$Original) clone(object);
         return new int$Pair(clone.num(), clone);
     }

     static public /*@ pure @*/ int$Pair den(IntRational$Original object) {
         IntRational$Original clone = (IntRational$Original) clone(object);
         return new int$Pair(clone.den(), clone);
     }

   /*@
     @ ensures (\forall IntRational$Original i; i != null;
               !true &&! (true && (true && true && true && true && true) &&
                (true && true && true && true && true) &&
                !(den(object).value() * den(other).value() == 0)) ||
                true && (true && (true && true && true && true && true) &&
                (true && true && true && true && true) &&
                !(den(object).value() * den(other).value() == 0)) &&
                equals(\result.value(), make(i, num(object).value() *
                num(other).value(), den(object).value() *
                                        den(other).value()).value()).value());
       @*/
     static public /*@ pure @*/ IntRational$Pair mult(
                  IntRational$Original object, IntRational$Original other) {
         IntRational$Original clone = (IntRational$Original) clone(object);
         return new IntRational$Pair(clone.mult(other), clone);
     }

     /*@
       @ requires true ==> !(!(true && true) && !true ||
                           (true && true) && true && (num(other).value() == 0));

 @ ensures !true && !(true && true && !(num(other).value() == 0) && true) ||
                 true && (true && true && !(num(other).value() == 0) && true) &&
                 equals(\result.value(), mult(object,
                                   inverse(other).value()).value()).value();
       @*/
     static public IntRational$Pair div(
                  IntRational$Original object, IntRational$Original other) {
         IntRational$Original clone = (IntRational$Original) clone(object);
         return new IntRational$Pair(clone.div(other), clone);
     }

     /*@
       @ requires true ==> !(!(true && true) && ! true ||
                           (true && true) && true && (num(object).value() == 0));

       @ ensures (\forall IntRational$Original i; i != null;
                 !true && !(true && (true && true) && (true && true) &&
                                            !(num(object).value() == 0)) ||
                 true && (true && (true && true) && (true && true) &&
                                            !(num(object).value() == 0)) &&
                 equals(\result.value(), make(i, den(object).value(),
                                    num(object).value()).value()).value());
       @*/
     static public /*@ pure @*/ IntRational$Pair inverse(
                                        IntRational$Original object) {
         IntRational$Original clone = (IntRational$Original) clone(object);
         return new IntRational$Pair(clone.inverse(), clone);
     }

     /*@
       @ ensures (\forall IntRational$Original r; r != null;
                 !true && !(true && true && true && !(1 == 0)) ||
                 true && (true && true && true && !(1 == 0)) &&
                 equals(\result.value(), make(r, 0, 1).value()).value());
       @*/
     static public /*@ pure @*/ IntRational$Pair zero(IntRational$Original object){
         IntRational$Original clone = (IntRational$Original) clone(object);
         return new IntRational$Pair(clone.zero(), clone);
     }

     /*@
       @ ensures \result.value() ==> other instanceof IntRational$Original &&
                 (!(true && true) && !(true && true) ||
                                            (true && true) && (true && true) &&
                 num(one).value() == num((IntRational$Original) other).value());

       @ ensures \result.value() ==> other instanceof IntRational$Original &&
                 (!(true && true) && !(true && true) ||
                                            (true && true) && (true && true) &&
                 den(one).value() == den((IntRational$Original) other).value());
       @ ensures other instanceof IntRational$Original && (
                  (true && (true && true) && (true && true)) ==
                  (true && (true && true) && (true && true))
                  &&
                  ! (true && (true && true) && (true && true)) &&
                  ! (true && (true && true) && (true && true)) ||
                   (true && (true && true) && (true && true)) &&
                   (true && (true && true) && (true && true)) &&
                    num(one).value() * den ((IntRational$Original) other).value()
                   == num((IntRational$Original) other).value() * den(one).value())
                  ==> \result.value();
       @*/
     static public /*@ pure @*/ boolean$Pair equals (
                                   IntRational$Original one, Object other) {
         IntRational$Original clone = (IntRational$Original) clone(one);
         return new boolean$Pair(clone.equals(other), clone);
     }

     /*@
       @ ensures equals(one, \result).value();
       @*/
     static public Object clone(IntRational$Original one) {
         return one.clone();
     }

 }
```

Fig. 26. Immutable class equipped with contracts