Ensuring Correctness of Java Frameworks: A Formal Look at JCF

Anna Mikhajlova

Turku Centre for Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14A, Turku 20520, Finland

Emil Sekerinski

McMaster University,
1280 Main Street West, Hamilton,
Ontario, Canada, L8S4K1

Turku Centre for Computer Science
TUCS Technical Report No 250
March 1999

ISBN 952-12-0402-8
ISSN 1239-1891

Abstract

In this paper we propose a novel approach to specification, development, and verification of object-oriented frameworks employing separate interface inheritance and implementation inheritance hierarchies. In particular, we illustrate how our method of framework specification and verification can be used to specify Java Collections Framework, which is a part of the standard Java Development Kit 2.0, and ensure its correctness. We propose to associate with Java interfaces formal descriptions of the behavior that classes implementing these interfaces and their subinterfaces must deliver. Verifying behavioral conformance of classes implementing given interfaces to the specifications integrated with these interfaces allows us to ensure correctness of the system.

The characteristic feature of our specification methodology is that the specification language used combines standard executable statements of the Java language with possibly nondeterministic specification statements. A specification of the intended behavior of a particular interface given in this language can serve as a precise documentation guiding implementation development. Since subtyping polymorphism in Java is based on interface inheritance, behavioral conformance of subinterfaces to their superinterfaces is essential for correctness of object substitutability in clients. As we view interfaces augmented with formal specifications as abstract classes, verifying behavioral conformance amounts to proving *class refinement* between specifications of superinterfaces and subinterfaces. Moreover, the logic framework that we use also allows verification of behavioral conformance between specifications of interfaces and classes implementing these interfaces. The uniform treatment of specifications and implementations and the relationships between them permits verifying correctness of the whole framework and its extensions.

Keywords: formal specification, reasoning, object-oriented frameworks, separate subtyping and subclassing, nondeterminism, correctness, verification, class refinement, Java

TUCS Research Group
Programming Methodology Research Group

1 Introduction

One of the main characteristic features of object-oriented frameworks is the extensibility, i.e., the ability of frameworks to call user extensions. In view of this important characteristic it is critical to build user extensions which *behaviorally conform* to the part of the framework that they extend. Moreover, frameworks themselves are usually build in a hierarchical manner, starting with a certain basic functionality and specializing this functionality in various directions to meet different demands. Naturally, behavioral conformance also underlies this hierarchy, with the most general behavior at the top level and specialized or refined behaviors at the lower levels. Verification of behavioral conformance both within a framework and between the framework and its extensions is critical for ensuring correctness and reliability of the resulting system.

In this paper we propose a specification and verification method supporting development of provably correct object-oriented frameworks. The method has been originally described in [19] in application to systems with unified interface and implementation inheritance hierarchies. Here we focus on object-oriented frameworks employing separate interface inheritance and implementation inheritance hierarchies and illustrate how our method of framework development can be used to specify Java Collections Framework (JCF) and ensure its correctness. Essentially, we propose to associate with Java interfaces formal descriptions of the behavior that classes implementing these interfaces and their subinterfaces must deliver. Interfaces always have an informal semantics as expressed in their names and in the names and parameter types of their methods, we just make this semantics explicit and express it mathematically. Such formal specifications can be distributed as part of the framework documentation, contributing to the detailed understanding of its functionality and guiding extension development. The characteristic feature of our specification methodology is that the specification language used combines standard executable statements of the Java language with possibly nondeterministic specification statements. Every statement in this language has a precise mathematical meaning in the refinement calculus as described in [19, 3]. In this paper we present only informal explanations of the specification constructs used in specifications of JCF interfaces.

Since subtyping polymorphism in Java is based on interface inheritance, behavioral conformance of subinterfaces to their superinterfaces is essential for correctness of object substitutability in clients. Our verification of behavioral conformance is based on the notion of *class refinement* first described in [19] and developed in [3]. One class (usually more abstract or nondeterministic) is refined by another class (usually more concrete or deterministic) if the externally observable behavior of the first class is preserved in the second class while decreasing nondeterminism. For a detailed description of refinement in the refinement calculus we refer to [21, 5]. Class refinement per se is based on data refinement [14, 12, 20, 4] which takes place when a state space is changed in a refinement step. An extensive collection of "high level" refinement laws that has been developed within the refinement calculus permits verification of class refinement in practice, and mechanical verification tools that are currently being developed [8] open the possibility of mechanized verification.

As we view interfaces augmented with formal specifications as abstract classes, verifying behavioral conformance amounts to proving class refinement between specifications of superinterfaces and subinterfaces. Moreover, the logical framework that we use also enables verification of behavioral conformance between specifications of interfaces and classes implementing these interfaces: class refinement must be established between the specification and the implementation classes. The uniform treatment of specifications and implementations and the relationships between them permits us to verify correctness of the original framework and then prove that user extensions preserve this correctness, ensuring in this way the correctness of the whole system.

The paper is organized as follows. In Sec. 2 we describe Java Collections Framework which we use to illustrate our approach, specify the interface *Collection* with its *Iterator* and the subinterface of *Collection*, *List*, with its *ListIterator*. Our specifications are entirely based on informal descriptions of the interface semantics as described in [6], and we reflect on the clarity and preciseness of these descriptions. In Sec. 3 we explain the notion of class refinement, present a number of refinement laws, and demonstrate verification of class refinement between the specifications of *Iterator* and *ListIterator*. Finally, in Sec. 4 we draw some conclusions, and describe future work.

Notation. We use *simply typed higher-order logic* as the logical framework in the paper. The type of functions from a type Σ to a type Γ is denoted by $\Sigma \to \Gamma$ and functions can have arguments and results of function type. Functions can be described using λ-abstraction and we write $f\ x$ for the application of function f to argument x. Whenever necessary to clarify the argument in the application of a function, especially in the case when the argument is a tuple of elements, we also use brackets around the argument, writing $f(x)$.

The use of equality and assignment symbols deserves special attention. The Java language uses $=$ to denote assignment and $==$ to denote equality of two values, and we will follow this convention in specifications. However, being reluctant to redefine the symbol $=$ traditionally used to denote mathematical equality, we will also use it in logical formulas and definitions, clarifying the intended meaning when necessary. In particular, we will use $=$ rather than $==$ to represent logical equality on right-hand sides of definitions and between the two parts of equational rules.

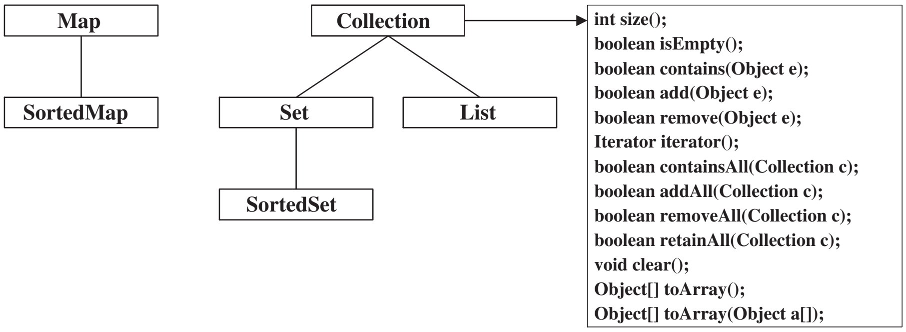

Figure 1: Collection hierarchy

2 Specifying Java Collections Framework

As was stated in the description of JCF [6], "A collections framework is a unified architecture for representing and manipulating collections." This particular framework contains three parts: interfaces, implementations, and algorithms. In this paper we focus on the interfaces, formalizing their informal descriptions as given in [6] and studying behavioral conformance between formal specifications of interfaces and formal specifications of their subinterfaces as we define them. The following description of JCF is based on [6].

The interfaces at the core of JCF form a hierarchy as shown in Fig. 1. The root of the hierarchy, the *Collection* interface, represents a group of objects, known as its elements. *Collection* is used to pass collections around and manipulate them when maximum generality is desired. Some *Collection* specializations allow duplicate elements and others do not. Some are ordered and others are not. For example, *Set* is an unordered collection that cannot contain duplicate elements, and *List* is an ordered collection that can contain duplicates.

2.1 Specifying the Collection Interface

In the *Collection* interface the method names suggest the intended functionality, for example, the method *size* returns the size of the underlying collection. The interface type *Iterator* returned by the method *iterator* is used to access collection elements and structurally modify the collection. In Fig. 2 we illustrate the hierarchy formed by *Iterator* and its subinterface *ListIterator*. The methods *hasNext*, *next*, and *remove* check whether there are more elements in the collection, return the next element, and remove the current element respectively. The description of JCF states that the behavior of an iterator is unspecified if the underlying collection is structurally modified while the iteration is in progress in any way other than by calling

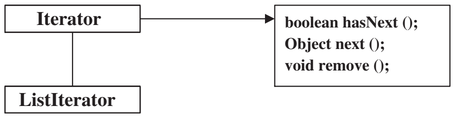

Figure 2: Iterator hierarchy

the method *remove*.

To specify the behavior of *Collection* methods we must model the underlying data structure the methods operate on. It appears to be rather natural to model this data structure by a bag (multiset) of *Object*[1] elements, as we want the collection to contain polymorphic elements, possibly duplicated or unordered. Furthermore, to specify the history of structural modifications, we will use an integer attribute *modified* which will be increased whenever elements are added to the original collection or removed from it. We begin with specifying the data attributes, the constructor, and the basic operations of *Collection* as follows:

public interface Collection {
  $bag\ of\ Object\ elems;$
  $int\ modified;$
  $Collection()$ {
    $elems, modified = [\![ ]\!], 0;$
  }
  int size() {
    return $min(\#elems,$ Integer.MAX_VALUE);
  }
  boolean isEmpty() {
    return $(\#elems == 0);$
  }
  boolean contains(Object o) {
    return $(o \in elems);$
  }
  boolean add(Object o) {
    $boolean\ r \mid r == false;$
    if $(o \in elems)\{$
      choose {skip; }
      or $\{elems, modified, r = elems + [\![o]\!], modified + 1, true;\}$;
    }
    else $\{elems, modified, r = elems + [\![o]\!], modified + 1, true;\}$;

[1]In Java the standard class *Object* is a superclass of all other classes, and a variable of type *Object* can hold a reference to an object of any other type.

```
    return r;
}
boolean remove(Object o) {
    boolean r | r == false;
    if (o ∈ elems){
        elems, modified = elems \ o, modified + 1;
        r = true;
    }
    return r;
}
Iterator iterator() {
    Iterator i = new Iterator(this);
    return i;
}
```

In this specification highlighted in bold is the original *Collection* interface and the rest is the precise description of the intended behavior. The behavior of the constructor and the methods is specified in terms of operations on bags and integers, with # returning the number of elements in a bag, $\in, +$, and $\setminus$ representing containment of an element in a bag, bag summation, and element removal respectively:

$$
\begin{array}{rcl}
\#b & \widehat{=} & \sum e \bullet b\ e \\
e \in b & \widehat{=} & b\ e > 0 \\
b_1 + b_2 & \widehat{=} & (\lambda e \bullet b_1\ e + b_2\ e) \\
b \setminus a & \widehat{=} & (\lambda e \bullet (e = a)\, ?\, \mathsf{max}\,((b\ e - 1),\, 0) : b\ e)
\end{array}
$$

As bags are functions from elements to the number of their occurrences, function application $b\ e$ returns the number of elements e in the bag b. In the last definition the equality on the right-hand side of definition sign $\widehat{=}$ is the logical equality. The conditional expression $b\,?\,e_1 : e_2$ is equal to the expression e_1 if the boolean condition b holds and to e_2 otherwise.

Finally, the statement choose S_1 or ... or S_n, used in the specification of the method *add*, represents a nondeterministic choice between the alternatives S_1 through S_n.[2]

Although the specifications of the constructor and the methods intuitively are quite straightforward, a few points are of interest here. First of all, assignment of a bag to a variable of type *bag of Object*, as in the constructor, results in the corresponding variable containing the value which

[2]Dijkstra's nondeterministic choice statement is usually written as $S_1 \,[\!]\, \ldots \,[\!]\, S_n$ or $S_1 \sqcap \ldots \sqcap S_n$, e.g. in [10, 5]. Here we use the syntax choose S_1 or ... or S_n instead because we believe that it improves readability.

is equal to the value being assigned, in this case $[\![\,]\!]$, with equality on bags defined as follows:

$$b1 == b2 \quad \widehat{=} \quad (\forall e \bullet b1\ e\ = b2\ e)$$

The description of method *contains* in [6] states that this method "returns true if and only if this Collection contains at least one element e such that $(o == null\ ?\ e == null\ :\ o.equals\,(e))$". Looking up the description of method *Object.equals*, we see that "for any reference values x and y, this method returns true if and only if x and y refer to the same object ($x == y$ has the value true)". Our specification states that the object reference o, be it a null or a non-null value, is one of the elements in the bag *elems*, which directly corresponds to the above description, still being more succinct and concise.

The description of method *add* states that it ensures that the current *Collection* instance contains the specified element, returning true if *Collection* changed as a result of the call and false if it does not permit duplicates and already contains the specified element. The nondeterministic choice operator `choose` used in our specification allows us to express these variations in the behavior succinctly and precisely: if the element to be added is already present in the current instance of *Collection*, this element can either be added to *Collection* or the addition of the element can be skipped, with the choice between the options made nondeterministically. When the element is not present, it is necessarily added to *Collection*. The declaration and initialization of a local variable r is equivalent to the declaration followed by assigning r the boolean value *false*. Further on in specifications we will use this kind of initialization along with nondeterministic initialization where a local variable is initialized according to some predicate.

The method *remove* is described as an operation removing an element e such that $(o == null\ ?\ e == null\ :\ o.equals\,(e))$, if *Collection* contains one or more such elements. Further it is stated in [6] that this method "returns true if the Collection contained the specified element (or equivalently, if the Collection changed as a result of the call)". In our specification we stipulate that if o is present in *Collection* at least once, its number of occurrences is decreased by one and the method returns true.

The iterator returned in the identically named method of the *Collection* interface is constructed by calling the constructor *Iterator* and passing it the reference to the current instance of *Collection*. Although *Iterator* is just an interface which cannot be used to produce instances, we provide its formal specification in the same way as for *Collection*, and by giving the specification of *Iterator*'s constructor, we define the precise meaning of its invocation in the method *Iterator* of *Collection*. An implementation of *Iterator* will have to define its own constructor, and an implementation of *Collection* will then return an instance created by this constructor in the method *Iterator*.

Figure 3: Simultaneous modification of a collection by different iterators

Before presenting a formal specification of the *Iterator* interface, let us consider a collaboration between a collection and iterators attached to it. As described in [6], "*Iterator.remove* is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress." Obviously, this description is rather ambiguous, because it is unclear the behavior of which methods is unspecified, and how modifications are being monitored, and what it means for an iteration to be in progress. To get an intuitive understanding of object interaction in this case, let us consider Fig. 3. Suppose that two iterators i_1 and i_2 are used to iterate over a collection implemented as a list, as shown in Fig. 3(a). Now, if we execute $i_2.next(); i_1.next(); i_1.remove()$, the iterator i_2 will be indexing a non-existing list element, as shown in Fig. 3(b). Further invocations of methods on i_2 will produce erroneous results or simply abort. However, the iterator i_1, which has carried out the structural modification of the underlying collection, will continue to work correctly. Accordingly, we have to specify the conditions under which iterators can be sure that the underlying data structure hasn't been structurally modified. The data attribute *modified* of *Collection* can be used for this purpose. Maintaining in *Iterator* an invariant that its own *modified* data attribute is equal to the one of the underlying *Collection*, helps solve the problem. Furthermore, the description of method *remove* states that this method can be called only once per call to *next*. To reflect this requirement in the specification, we maintain a data attribute *canRemove* and set it to true after resetting the next element and to false after removing the current element. The interface *Iterator* can, therefore, be specified as follows:

public interface Iterator {

Collection col;

bag of Object current;

boolean canRemove;

int modified;

Object next;

invariant I $==$ $col\ != null \wedge$
$(canRemove \Rightarrow next \in current)$

interclass invariant $intI$ $==$ $current \subseteq col.elems \wedge$
$modified == col.modified$

```
    Iterator Collection c) {
      assert c != null;
      col, current, canRemove, modified, next = c, ⟦⟧, false, c.modified, null;
    }
    boolean hasNext() {
      return current ⊂ col.elems;
    }
    Object next() {
      assert current ⊂ col.elems;
      [next = e | e ∈ (col.elems \ current)];
      current, canRemove = current + ⟦next⟧, true;
      return next;
    }
    void remove() {
      assert canRemove;
      col.elems, col.modified = col.elems \ next, col.modified + 1;
      current, canRemove, modified = current \ next, false, modified + 1;
      next = null;
    }
  }
```

As elements in a bag cannot be indexed, we use the data attribute *current* to store the elements of the underlying collection that have been returned by the method *next* in the current iteration. The attribute *next* stores the element returned by the last call to the method *next*. The class invariant I states that the iterator is always attached to an existing collection ($col \mathrel{!=} null$) and that the next element to be removed is one of the elements currently "indexed" ($canRemove \Rightarrow next \in current$). This class invariant holds of all *Iterator* instances during their whole life cycle, being established by the constructor and preserved by all the methods. Apart from the class invariant, *Iterator* maintains another invariant *intI* which captures the invariance in relation between the attributes of *Iterator* and the attributes of *Collection* that it aggregates, stating that the elements returned by the method *next* are always in the underlying collection ($current \subseteq col.elems$) and that structural modifications made so far have been made by the current instance of *Iterator* ($modified == col.modified$). This invariant is different from the class invariant proper in that it is maintained mutually by *Iterator* and *Collection*. We choose to call this invariant "interclass invariant" to reflect that, on the one hand, it is an invariant established by the constructor and preserved by all the methods of *Iterator*, and, on the other hand, it is the predicate which cannot be assumed to hold of all *Iterator* instances at all times because *Iterator* alone cannot guarantee its preservation between method calls to its methods. In other words, creating an instance of *Iterator* through calling the constructor establishes *intI*, and, although there are no guarantees that *intI* holds at all moments in a life cycle of this instance, if it does then a call to any method of *Iterator* will preserve it. Note that the methods *add* and *remove* of *Collection* break *intI* which suggests potential behavioral problems with structural modification of the underlying collection by different iterators. The interclass invariant of a particular *Iterator* instance will be preserved only if this instance is used by the underlying collection to structurally modify itself through calls to *Iterator* methods. In this respect, the fact that *Collection* has the method *add*, while *Iterator* does not, might indicate the possibility of inadequate framework design.

Bertrand Meyer in [17] discusses the problem of interclass invariants, although in a slightly different setting with two classes maintaining mutual references to each other, and proposes to do run-time monitoring of these invariants, effectively adding them to pre- and post-conditions of methods in the classes whose attributes are related through such invariants. We define the semantics of the interclass invariant construct similarly, by adding it as the implicit assert condition in the end of the class constructor and the implicit assume\assert conditions in, respectively, the beginning and the end of every class method. Proving consistency of a class with respect to its class invariant and interclass invariant amounts to verifying that both kinds of invariants are established by the class constructor and preserved by all its methods.

The additional operations on bags used in the specification of *Iterator* are defined as follows:

$$\begin{array}{lcl} b_1 \subseteq b_2 & \widehat{=} & (\forall e \bullet b_1\ e \leq b_2\ e) \\ b_1 \subset b_2 & \widehat{=} & b_1 \subseteq b_2 \wedge \#b_1 < \#b_2 \\ b_1 \setminus b_2 & \widehat{=} & (\lambda e \bullet \mathsf{max}\,((b_1\ e - b_2\ e),\, 0)) \end{array}$$

Apart from standard Java language constructs we use the multiple assignment statement $x_1, \ldots, x_n = e_1, \ldots, e_n$ which stands for a simultaneous assignment of expressions $e_1, \ldots, e_n$ to variables $x_1, \ldots, x_n$ respectively. Assuming that $x_1, \ldots, x_n$ do not occur free in $e_1, \ldots, e_n$, multiple assignment can always be rewritten as a sequential composition of the corresponding individual assignments in arbitrary order. Moreover, we use two specification statements, assertion and nondeterministic update. The assertion statement $\mathsf{assert}\ p$, where p is a boolean-valued expression, skips if p holds in a current state and aborts otherwise.[3] The nondeterministic update $[x = x' \mid b]$ assigns x a value x' satisfying a boolean condition b; if such a value cannot be found, the execution stops.

The assertion $\mathsf{assert}\ c\ != \mathit{null}$ stipulates that the constructor creates a new *Iterator* instance only under the condition that the collection referred

[3] The syntax of the assertion statement is different in [5] where the semantics of this statement is defined; it is written as $\{p\}$ instead of $\mathsf{assert}\ p$ that we use here. Using the syntax $\{p\}$ to denote assertions in Java specifications would be confusing, as the curly brackets are used to delineate blocks.

by c is some existing object, otherwise the constructor aborts. The method *next* returns a next object in the underlying data structure only under the condition that the end of the structure hasn't been reached, as expressed in the assertion assert $current \subset col.elems$. Note that the element to be returned by this method is chosen nondeterministically from the elements in the underlying collection that haven't been returned by *next* in the current iteration run. This element is added to the bag of currently iterated elements *current* and the boolean flag *canRemove* is set to true, permitting removal of the next element. In turn, the method *remove* agrees to remove the next element only if *canRemove* holds in a state, encoding the requirement that *remove* can be called only once per call to *next*.

Note that in the specification of *Iterator* we directly modify data attributes of the aggregated collection *col*. Normally, in object-oriented programming such practice is rightfully criticized for breaking encapsulation and is recommended against. In specifications, however, we will permit such direct access and modification because this significantly simplifies specifications, as there is no need to specify the behavior solely in terms of method calls on the aggregated objects. There is no danger of breaking encapsulation because implementations can (and usually will) use completely different attributes for achieving what is required in the specification, and in the implementations direct access to data attributes of another class will be completely eliminated and substituted with method calls preserving encapsulation.

Now we can continue with specifying the bulk operations of *Collection* as follows:

```
public interface Collection {
  ...
  boolean containsAll(Collection c) {
    assert c != null;
    return c.elems ⊆ elems;
  }
  boolean addAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟦⟧);
    bag of Object old, int cm | old == elems ∧ cm == c.modified;
    [elems, modified = e, m | e == elems + c.elems ∧
      m ≥ modified ∧ c.modified == cm];
    return old != elems;
  }
  boolean removeAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟦⟧);
    boolean r, int cm | r == false ∧ cm == c.modified;
    if (∃e • e ∈ c.elems ∧ e ∈ elems) {
```

$[elems, modified = e, m \mid e == elems \setminus toSet(c.elems)\ \wedge$
$\quad m \geq modified\ \wedge\ c.modified == cm];$
$r = true;$
};
return r;
}
boolean retainAll(**Collection c**) {
assert c != $null\ \wedge\ (c ==$ this $\Rightarrow c.elems == [\![\,]\!]);$
$boolean\ r, int\ cm \mid r == false\ \wedge\ cm == c.modified;$
if $(\exists e \bullet e \in c.elems \wedge e \in elems)$ {
$[elems, modified = e, m \mid e == elems \setminus toSet(elems \setminus toSet(c.elems))\ \wedge$
$\quad m \geq modified\ \wedge\ c.modified == cm];$
$r = true;$
};
return r;
}
void clear() {
$elems = [\![\,]\!];$
$[modified = m \mid m \geq modified];$
}
}

The specification of method *containsAll* is quite straightforward: under the condition that the reference to the incoming *Collection* is non-null this method returns true if all elements in the incoming *Collection* are present in the current instance of *Collection*. Note that in the specification it is assumed that the incoming *Collection* is an instance of the specification class *Collection* whose attribute *elems* is a bag of *Object* elements. The behavior of this method in the case when an instance of some other class implementing the interface *Collection* is passed as input is underspecified. The implementation of *containsAll* will have to be polymorphic and deliver the behavior as specified in *Collection.containsAll* regardless of the dynamic type of the input argument.

The informal description of method *addAll* states that the behavior of this operation is undefined if the incoming *Collection* is modified while the operation is in progress. To express this restriction in the specification, we use the local variable *cm* to keep the number of modifications made to c up to the moment it was passed as input to *addAll*. Elements of c are guaranteed to be added to the current *Collection* only if *cm* remains equal to *c.modified* during the whole operation. Also it is mentioned in [6] that the behavior of *addAll* is undefined if the incoming *Collection* is the current instance of *Collection* and is nonempty. We address this restriction by stating the corresponding assertion in the beginning of the method specification.

Note how this specification of *addAll* uses specification constructs to express the required complex functionality. On the one hand, we avoid unnecessary details, such as checking whether the current *Collection* gets modified as a result of each call to *add* and simply return the result of comparing the original bag with the resulting one. This specification is inefficient but it succinctly and clearly captures the intended behavior. On the other hand, we do not oversimplify the specification sacrificing preciseness: writing just $elems = elems + c.elems$ would certainly make the specification of this method shorter, but wouldn't express the necessary requirement that the collection c cannot be modified during the addition. An implementation of *addAll* will add elements iteratively, and in order to meet the requirement about non-modification of c it will have to check that this requirement is satisfied before adding each element of c.

Surprisingly, the informal description of method *removeAll* does not stipulate the requirement that the incoming collection c cannot be modified during its execution. However, based on the statement that "After this call returns, this *Collection* will contain no elements in common with the specified *Collection*" we can justify the need for such a requirement. Suppose that, while iterating over c, an element which already has been removed from the current *Collection* is added again to c before the iterator used in *removeAll*. When the execution of *removeAll* completes, the current *Collection* will still have some elements in common with c. Similarly the result of *removeAll* can be undefined if some elements of c are externally removed during the iteration. It is also easy to see that the behavior of this method becomes undefined if the current *Collection* is passed to it as an input argument. In the specification we address all these requirements using the corresponding assertions and the nondeterministic assignment statement. The latter states that the new value assigned to *elems* is equal to the difference between the current *Collection* and the set obtained from converting the bag of elements in c; in addition, it is stipulated that c does not undergo any structural modifications: its *modified* attribute remains unchanged. The function *toSet* used in this specification is defined for a bag b as follows:

$$toSet(b) \quad \widehat{=} \quad \{e \mid b\ e > 0\}$$

The function returning the difference between a bag b and a set s is given as follows:

$$b \setminus s \quad \widehat{=} \quad (\lambda e \bullet (e \in s)\,?\,0 : b\ e)$$

In the specification of method *retainAll* we state that the new value assigned to *elems* contains only the elements that are common to the original bag *elems* and the incoming *c.elems*. The same non-modification requirements as in *removeAll* are imposed on c for similar reasons. Finally, the method *clear* results in assigning to *elems* the empty bag $[\![\,]\!]$.

The next two methods to be specified deal with converting *Collection* to an array. The first method *toArray* is described in [6] as one returning an array containing all of the elements in the current *Collection*. It is stated that "the returned array will be 'safe' in that no references to it are maintained by *Collection*". As such this is a rather vague description of the behavior, because it is unclear whether elements of the original collection are copied to the returned array by reference or by value. When writing a formal specification we must address this issue, and we choose to copy the collection elements by value rather than by reference, which appears to be safer than copying by reference.

The behavior of the second *toArray* method is described as follows: "Returns an array containing all of the elements in this *Collection*, whose runtime type is that of the specified array. If *Collection* fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this *Collection*. If *Collection* fits in the specified array with room to spare (i.e., the array has more elements than *Collection*), the element in the array immediately following the end of the collection is set to null." We specify the two array conversion methods as follows:

public interface Collection {

$\ldots$

Object[] **toArray**() {

$Object[]a = \mathsf{new}\ Object[\#elems];$

$bag\ of\ Object\ be \mid be == elems;$

$\mathsf{for}\ (i = 0; i < \#elems; i = i + 1)\ \{$

$[a[i], be = a, b \mid a \notin be\ \wedge\ (\exists a' \bullet a' \in be\ \wedge\ a\!\uparrow == a'\!\uparrow \wedge b == be \setminus a']$

$\};$

$\mathsf{return}\ a;$

}

Object[] **toArray**(**Object a**[]) {

$Class\ typeOfArray = a.getClass().getComponentType();$

$bag\ of\ Object\ be \mid be == elems;$

$\mathsf{if}\ (a.length() < \#elems)\ \{$

$Object[]c = \mathsf{new}\ typeOfArray[\#elems];$

$\mathsf{for}\ (i = 0; i < \#elems; i = i + 1)\ \{$

$[c[i], be = c, b \mid c \notin be\ \wedge\ (\exists c' \bullet c' \in be\ \wedge\ c\!\uparrow == c'\!\uparrow \wedge$

$b == be \setminus c'\ \wedge\ c.getClass() == typeOfArray)]$

$\};$

$\mathsf{return}\ c;$

$\}$

$\mathsf{else}\ \ \{$

$\mathsf{for}\ (i = 0; i < \#elems; i = i + 1)\ \{$

$$[a[i], be = a, b \mid a \notin be \wedge (\exists a' \bullet a' \in be \wedge a\!\uparrow == a'\!\uparrow \wedge$$
$$b == be \setminus a' \wedge a.getClass() == typeOfArray)]$$

};
$a[\#elems] = null$;
return a;
};
}
}

One interesting point to note here is the use of method invocations *getClass* and *getComponentType*. Although the precise definitions of these methods, supported by the array interface, are not available, we include these method invocations in the specification of method *toArray* to indicate that these methods should be called in implementations of *Collection*. Being partial, such a specification of the behavior of *toArray* is nevertheless very useful, as it succinctly describes the intended actions and guides implementation development.

This concludes our specification of *Collection* and now we can specify the interface *List* which extends *Collection*.

2.2 Specifying List and ListIterator

A *List* is an ordered *Collection* sometimes called a sequence. In addition to the operations inherited from *Collection*, the interface *List* includes operations for positional access, search for a specified object in the list, list iteration, and range operations on the list. In addition to the ordinary *Iterator*, *List* provides a richer *ListIterator* that allows one to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator. The interfaces of *List* and *ListIterator* are shown in Fig. 4.

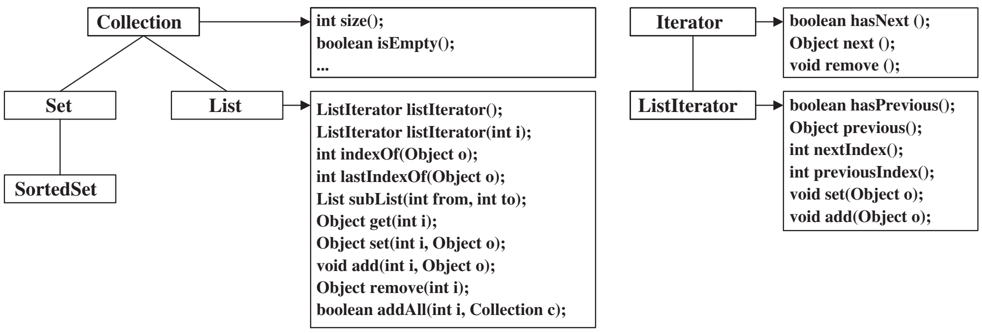

Figure 4: List and ListIterator interfaces

It appears to be natural to model the underlying data structure of *List* by a sequence of *Object* elements. As before, to specify the history of structural modifications, we will use an integer attribute *modified* which will be increased whenever elements are added to the original list or removed from it. We begin with specifying the data attributes, the constructor, and the operations inherited from *Collection*:

public interface List extends Collection {

$\quad seq\ of\ Object\ elems;$

$\quad int\ modified;$

$\quad List()\ \{$

$\qquad elems, modified = \langle\rangle, 0;$

$\quad \}$

$\quad$ **int size**() {

$\qquad$ return $min(\#elems, \text{Integer.MAX\_VALUE});$

$\quad \}$

$\quad$ **boolean isEmpty**() {

$\qquad$ return $(\#elems == 0);$

$\quad \}$

$\quad$ **boolean contains(Object o)** {

$\qquad$ return $(o \text{ in } elems);$

$\quad \}$

$\quad$ **boolean add(Object o)** {

$\qquad elems, modified = elems ^\frown \langle o \rangle, modified + 1;$

$\qquad$ return $true;$

$\quad \}$

$\quad$ **boolean remove(Object o)** {

$\qquad boolean\ r \mid r == false;$

$\qquad$ if $(o \text{ in } elems)\{$

$\qquad\quad [elems = e \mid (\exists l_1, l_2 \bullet$

$\qquad\qquad l_1 ^\frown \langle o \rangle ^\frown l_2 = elems \wedge \neg(o \text{ in } l_1) \wedge l_1 ^\frown l_2 = e)];$

$\qquad\quad modified = modified + 1;$

$\qquad\quad r = true;$

$\qquad \}$

$\qquad$ return $r;$

$\quad \}$

$\quad$ **Iterator iterator**() {

$\qquad Iterator\ i = \text{new } ListIterator(\text{this});$

$\qquad$ return $i;$

$\quad \}$

Specifications of the constructor and the methods *size*, *isEmpty*, and *contains* are quite straightforward with the membership operation in on sequences defined as follows:

$$e \text{ in } l \quad \widehat{=} \quad (\exists i \mid 0 \leq i < \#l \bullet l[i] = e)$$

To improve readability we use the notation $l[i]$ rather than the function application $l\ i$ to represent the i'th element of the sequence l.

The method *add* appends the specified element to the end of *List*, whereas the method *remove* removes the first occurrence of the specified element from *List*. The iterator returned by the identically named method is an instance of *ListIterator* which is an extension of *Iterator*. In addition to the methods of *Iterator*, *ListIterator* provides methods allowing positional access through an index.

The bulk operations and the array conversion operations of *List* are specified similarly to those of *Collection*:

```
public interface List extends Collection {
  ...
  boolean containsAll(Collection c) {
    assert c != null;
    return c.elems ⊆ toBag(elems);
  }
  boolean addAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟨⟩);
    seq of Object old, int cm | old == elems ∧ cm == c.modified;
    [elems, modified = e, m | ∃e' •
      toBag(e') == c.elems ∧ e == elems⌢e' ∧
      m ≥ modified ∧ c.modified == cm];
    return old != elems;
  }
  boolean removeAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟨⟩);
    boolean r, int cm | r == false ∧ cm == c.modified;
    if (∃e • e ∈ c.elems ∧ e in elems) {
      [elems, modified = e, m | e == remAll(elems, toSet(c.elems)) ∧
        m ≥ modified ∧ c.modified == cm];
      r = true;
    };
    return r;
  }
  boolean retainAll(Collection c) {
    assert c != null ∧ (c == this ⇒ c.elems == ⟨⟩);
```

```
      boolean r, int cm | r == false ∧ cm == c.modified;
      if (∃e • e ∈ c.elems ∧ e ∈ elems) {
        [elems, modified = e, m |
          e == remAll(elems, toSet(elems) \ toSet(c.elems)) ∧
          m ≥ modified ∧ c.modified == cm];
        r = true;
      };
      return r;
    }
    void clear() {
      elems = ⟨⟩;
      [modified = m | m ≥ modified];
    }
  }
```

The specifications of array conversion methods are similar to those of *Collection* and we omit them for the lack of space. The function *toBag* used in the specifications of methods *containsAll* and *addAll* is given as follows:

$$toBag(l) \;\widehat{=}\; \begin{cases} [\![\,]\!] & \text{if } l = \langle\rangle \\ toBag(front) + [\![e]\!] & \text{if } l = front \frown \langle e \rangle \end{cases}$$

The function *remAll* used in the specifications of methods *removeAll* and *retainAll* is given as follows:

$$remAll(l, s) \;\widehat{=}\; \begin{cases} \langle\rangle & \text{if } l = \langle\rangle \\ remAll(front, s) & \text{if } l = front \frown \langle e \rangle \wedge e \in s \\ remAll(front, s) \frown \langle e \rangle & \text{if } l = front \frown \langle e \rangle \wedge e \notin s \end{cases}$$

Before proceeding with the specification of the new methods of *List*, let us present the specification of *ListIterator*. In this specification we use $l[f..t]$ to denote a subsequence of the given sequence l between indices f and t inclusive. We define this function to be total, returning a subsequence starting at index f and ending at index t, if these indices are such that $0 \leq f < t < \#l$, returning part of the sequence within range $f..\#l-1$ if the lower index f satisfies $0 \leq f < t$ but the upper index t is greater than $\#l-1$, and returning an empty sequence if the indices f and t are misplaced in some way, being in the wrong order or reaching outside the bounds $0..\#l-1$.

$$l[f..t] \;\widehat{=}\; \begin{cases} \langle l[t] \rangle & \text{if } 0 \leq f = t < \#l \\ l[f..t-1] \frown \langle l[t] \rangle & \text{if } 0 \leq f < t < \#l \\ l[f..\#l-1] & \text{if } t \geq \#l - 1 \\ \langle\rangle & \text{otherwise} \end{cases}$$

The specification of *ListIterator* can now be given as follows:

public interface ListIterator extends Iterator {

$\quad List\ lst;$

$\quad int\ ind;$

$\quad boolean\ canModify;$

$\quad int\ modified;$

$\quad$ invariant $J \quad == \quad lst\ != null$

$\quad$ interclass invariant $intJ \quad == \quad -1 \leq ind \leq \#lst.elems \wedge$
$(canModify \Rightarrow 0 \leq ind < \#lst.elems) \wedge$
$modified = lst.modified$

$\quad ListIterator(List\ l)\ \{$

$\qquad$ assert $l\ != null;$

$\qquad lst, ind, modified, canModify = l, -1, l.modified, false;$

$\quad \}$

$\quad$ **boolean hasNext**() {
$\qquad$ return $ind < \#lst.elems - 1;$
$\quad \}$

$\quad$ **boolean hasPrevious**() {
$\qquad$ return $ind > 0;$
$\quad \}$

$\quad$ **Object next**() {
$\qquad$ assert $ind < \#lst.elems - 1;$
$\qquad ind, canModify = ind + 1, true;$
$\qquad$ return $lst.elems[ind];$
$\quad \}$

$\quad$ **Object previous**() {
$\qquad$ assert $ind > 0;$
$\qquad ind, canModify = ind - 1, true;$
$\qquad$ return $lst.elems[ind];$
$\quad \}$

$\quad$ **int nextIndex**() {
$\qquad$ return min $(ind + 1, \#lst.elems);$
$\quad \}$

$\quad$ **int previousIndex**() {
$\qquad$ return max $(ind - 1, -1);$
$\quad \}$

$\quad$ **void remove**() {

$\qquad$ assert $canModify;$

$\qquad lst.elems = lst.elems[0..ind - 1]^\frown$

$\qquad\quad lst.elems[ind + 1..\#lst.elems - 1];$

$\qquad ind, modified, canModify = ind - 1, modified + 1, false;$

$\qquad lst.modified = modified;$

$\quad \}$

$\quad$ **void set(Object o)** {

$\qquad$ assert $canModify;$

$\qquad lst.elems[ind] = o;$

$\quad \}$

$\quad$ **void add(Object o)** {

$\qquad ind = ind + 1;$

$\qquad [lst.elems = s \mid \exists s_1, s_2 \bullet s == s_1{}^\frown \langle o \rangle{}^\frown s_2\ \wedge$

$\qquad\quad lst.elems == s_1{}^\frown s_2\ \wedge\ s[ind] == o];$

$\qquad canModify, modified = false, modified + 1;$

$\qquad lst.modified = modified;$

$\quad \}$

}

Just as *Iterator* is used to iterate over its aggregated *Collection*, *ListIterator* is used to iterate over *List*. The class invariant J of *ListIterator* states that at all times it aggregates a non-null reference to a *List* instance. In addition, the interclass invariant *intJ* states that the integer-valued index *ind* is used to iterate over $lst.elems$ and can range in the interval $[-1..\#lst.elems]$, with valid index values being in the interval $[0..\#lst.elems - 1]$. The data field *canModify* is similar to *canRemove* of *Iterator* and is used to regulate the order of calls to *next* and *previous* before calls to $remove, add$, and *set*. Finally, the data field *modified* is used to regulate structural modifications made to the underlying list.

In the description of *List* interface in [6] the index is said to always be between two elements, the one that would be returned by a call to *previous* and the one that would be returned by a call to *next*. With this layout the index has $n + 1$ valid positions for the list of size n, starting with 0 and ending with n. In our opinion this intuitive picture is somewhat confusing, especially in the two boundary cases when the index is before the first element or past the last one. In fact, this layout is so confusing that we have found contradicting descriptions of method behavior. For example, in the section describing the interface *List* the method *nextIndex* is said to return $list.size() + 1$ when the cursor is after the final element, whereas in the documentation describing *ListIterator* proper it is stated that this method "returns list size if the list iterator is at the end of the list". Apparently, the confusion arises because of the ambiguity of the valid values of the index pointing between elements rather than at elements. With our specification the index positions -1 and $\#lst.elems$ are boundary, whereas if the index *ind* is in the interval $[0..\#lst.elems - 1]$ inclusive, it points to the elements $lst.elems[0]$ through $lst.elems[\#lst.elems - 1]$. Having decided on the relationship between the index and the list elements, we can specify the behavior of *ListIterator* constructor and methods unambiguously. Namely, in the constructor the index is set to the boundary position -1. The methods *next* and *previous* first check that moving the index to the next (previous) position would not take it outside the bounds, then increment (decrement) it and return the currently indexed list element. The method *nextIndex* returns the minimum between $ind + 1$ and the size of the list, whereas the method *previousIndex* returns the maximum between $ind - 1$ and the boundary value -1. Obviously, the specifications of these methods are not only unambiguous but also very concise.

The specifications of methods *remove* and *set* are quite straightforward but the specification of *add* is worthy of a few comments. Let us first consider it description in [6]: "The element is inserted immediately before the next element that would be returned by next, if any, and after the next element that would be returned by previous, if any. (If the list contains no elements, the new element becomes the sole element on the list.) The new element is inserted before the implicit index: a subsequent call to next would be unaffected, and a subsequent call to previous would return the new element. (This call increases by one the value that would be returned by a call to nextIndex or previousIndex.)" The first sentence in this description is somewhat equivocal because it is unclear whether the element is inserted before the next element that would be returned by *next* if the inserted element wouldn't have been inserted, or it is inserted before the next element that would be returned by *next* if we call *next* after a call to *add*. In the first case, the result of *add* should be insertion of the new element into the position after the implicit index, whereas in the second case, the element returned by the method *next* depends not only on the position where the new element is inserted but also on the position where the implicit index is placed as the effect of *add*. Only the following sentences clarify that the intention is to place the new element into the position "before the implicit index".

Now that we know the exact behavior of *ListIterator*, we can proceed with specification of *List* operations for positional access, search, and range extraction.

```
public interface List extends Collection {
  ...
  ListIterator listIterator() {
    ListIterator itr = new ListIterator(this);
    return itr;
  }
  ListIterator listIterator(int i) {
    assert 0 ≤ i ≤ #elems;
    ListIterator itr = new ListIterator(this);
    itr.ind = i;
    return itr;
  }
  int indexOf(Object o) {
    return min ({i | elems[i] == o} ∪ {−1});
  }
  int lastIndexOf(Object o) {
    return max ({i | elems[i] == o} ∪ {−1});
  }
  List subList(int from, int to) {
    assert 0 ≤ from ≤ to ≤ #elems;
    seq of Object s | (∀i | from ≤ i < to • elems[i] == s[i − from]);
    List sub = new List();
    sub.elems, sub.modified = s, 0;
    return sub;
  }
```

Object get(**int i**) {
assert $0 \le i < \#elems$;
return $elems[i]$;
}
Object set(**int i**, **Object o**) {
assert $0 \le i < \#elems$;
$Object\ s \mid s == elems[i]$;
$[elems = e \mid \exists s_1, s_2 \bullet elems == s_1 {}^\frown \langle s \rangle {}^\frown s_2 \wedge \#s_1 == i \wedge e == s_1 {}^\frown \langle o \rangle {}^\frown s_2]$;
return s;
}
void add(**int i**, **Object o**) {
assert $0 \le i \le \#elems$;
$[elems = e \mid \exists s_1, s_2 \bullet elems == s_1 {}^\frown s_2 \wedge \#s_1 == i \wedge e == s_1 {}^\frown \langle o \rangle {}^\frown s_2]$;
$modified = modified + 1$;
}
Object remove(**int i**) {
assert $0 \le i < \#elems$;
$Object\ o \mid o == elems[i]$;
$[elems = e \mid \exists s_1, s_2 \bullet elems == s_1 {}^\frown \langle o \rangle {}^\frown s_2 \wedge \#s_1 == i \wedge e == s_1 {}^\frown s_2]$;
$modified = modified + 1$;
return o;
}
boolean addAll(**int i**, **Collection c**) {
assert $0 \le i \le \#elems \wedge (c ==$ this $\Rightarrow c.elems == [\![\,]\!])$;
$Iterator\ itr = c.iterator()$;
$int\ cm, seq\ of\ Object\ s, old \mid s == \langle\rangle \wedge old == elems \wedge cm == c.modified$;
while $(itr.hasNext())$ $\{s = s {}^\frown itr.next();\}$;
$[elems, modified = e, m \mid (\exists s_1, s_2 \bullet elems == s_1 {}^\frown s_2 \wedge \#s_1 == i \wedge$
$e == s_1 {}^\frown s {}^\frown s_2) \wedge m \ge modified \wedge c.modified == cm]$;
return old != $elems$;
}
}

The first two methods construct new *ListIterator* instances, setting their indices to -1 and the specified index i respectively. The next two methods *indexOf* and *lastIndexOf* return the indices of the first and the last occurrence of the specified element in the current *List*, or -1 if it does not contain this element. We specify these methods by saying that the returned index is, respectively, the minimal and the maximal element of the set containing all indices at which the list element is equal to the specified element or -1, if this set is empty. The description of method *subList*, whose specification is given next, states that the returned list is a portion of the current *List* between the specified *from* index, inclusive, and *to* index, exclusive. We specify this behavior by constructing a subsequence s of *elems* such that the elements of s are equal to the elements of *elems* starting at index *from* and finishing at index $to - 1$. A new *List* instance initialized with this subsequence is then returned as the result of method *subList*. The specifications of methods *set*, *add*, and *remove* are rather straightforward and hardly require further explanation. The behavior of method *addAll* adding the specified *Collection* at a specified position is very similar to the ordinary *addAll*. The only interesting point here is that the informal description of this method stipulates that the new elements will appear in the current *List* in the order that they are returned by the specified *Collection*'s iterator. We address this requirement by iteratively constructing from *Collection* elements a sequence and adding this sequence at the specified position in the current list.

3 Ensuring Correctness of JCF

As was already mentioned in the introduction, correctness of a framework can be ensured by verifying behavioral conformance between classes whose instances are intended for polymorphic substitution in clients. In systems with separate interface inheritance and implementation inheritance hierarchies, such as JCF, subtyping polymorphism is based on interface inheritance. Therefore, there are two ways of achieving polymorphic reuse, through passing instances of classes implementing an interface where objects with this interface are expected and through substituting objects of subinterface type for objects of superinterface type. In the first case, the concrete class must be shown to refine the specification of the interface it implements. In the second case, verifying behavioral conformance between the superinterface objects and the subinterface objects amounts to proving class refinement between the specification of the original interface and the specification of its subinterface. These two cases are illustrated in Fig. 5. The classes *AbstractCollection*, *ConcreteCollection* and *SpecialCollection* are different implementations of *Collection* interface and the classes *AbstractList* and *LinkedList* are different implementations of *List* interface. Both *Collection* and *List* interfaces are augmented with formal specifications of the in-

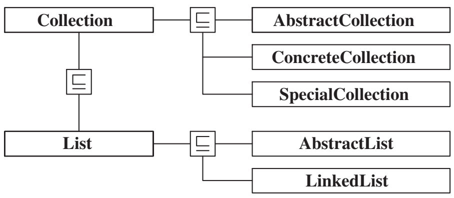

Figure 5: Behavioral conformance in systems with separate interface inheritance and implementation inheritance hierarchies

tended behavior. If we verify that the specification of *Collection* is refined by its implementations, i.e. if we prove class refinements *Collection* $\sqsubseteq$ *AbstractCollection*, *Collection* $\sqsubseteq$ *ConcreteCollection* and *Collection* $\sqsubseteq$ *SpecialCollection*, then clients specified to work with a variable c of type *Collection* will continue to work correctly if c is assigned an instance of any of the classes *AbstractCollection*, *ConcreteCollection* and *SpecialCollection*. Similarly, using an instance of *AbstractList* or *LinkedList* in the context where it is viewed as an object of type *List* will be correct if *List* $\sqsubseteq$ *AbstractList* and *List* $\sqsubseteq$ *LinkedList*.

Moreover, since *List* is a subinterface of *Collection*, instances of *AbstractList* and *LinkedList* can be used in the context where an object of type *Collection* is expected. If we verify that *Collection* $\sqsubseteq$ *List* then by transitivity *AbstractList* and *LinkedList*, as well as all other correct implementations of *List*, will be refinements of *Collection*.

We will illustrate the verification of class refinement by proving that *Iterator* is refined by *ListIterator*. But first we would like to explain the notion of refinement is more detail. For a formal treatment of class refinement we refer to [19, 3, 2].

3.1 Formal Background: Class Refinement

3.1.1 Semantics, Correctness and Refinement of Program Statements

Every program statement has a weakest precondition predicate transformer semantics. A *predicate transformer* $S : (\Gamma \to Bool) \to (\Sigma \to Bool)$ is a function from predicates on Γ to predicates on Σ. We write

$$\Sigma \mapsto \Gamma \ \widehat{=}\ (\Gamma \to Bool) \to (\Sigma \to Bool)$$

to denote the type of all predicate transformers from Σ to Γ. A statement with initial state in Σ and final state in Γ determines a monotonic predicate transformer $S : \Sigma \mapsto \Gamma$ that maps any postcondition state predicate $q : \Gamma \to Bool$ to the weakest precondition state predicate $p : \Sigma \to Bool$ such that the statement is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p. In the refinement calculus program statements are identified with the monotonic predicate transformers that they determine. For details of the predicate transformer semantics, we refer to [5].

The *total correctness assertion* $p \ \{\!|\ S\ |\!\}\ q$ is said to hold if the statement S can be used to establish the postcondition q when starting in the set of states p. Formally, the total correctness assertion $p \ \{\!|\ S\ |\!\}\ q$ is defined to be equal to $p \subseteq S\ q$, which means that p is stronger than the weakest precondition of S with respect to q.

A statement S is *refined by* a statement S', written $S \sqsubseteq S'$, if any condition that we can establish with the first statement can also be established with the second statement. Formally, $S \sqsubseteq S'$ is defined to hold if $p \; \{\!| \, S \, |\!\} \; q \Rightarrow p \; \{\!| \, S' \, |\!\} \; q$, for any p and q. Refinement is reflexive and transitive.

The refinement calculus provides rules for transforming more abstract program structures into more concrete ones based on the notion of refinement of statements presented above. For example, we have the following law for assignment introduction:

$$\mathsf{assert}\ b_1; [x = x' \mid b_2] \quad \sqsubseteq \quad x = e, \ \text{ if } b_1 \ \Rightarrow \ b_2[x' \leftarrow e] \tag{1}$$

This law states that a nondeterministic assignment to x a new value satisfying the boolean expression b_2 under the condition that b_1 holds initially is refined by a deterministic assignment of an expression e to x if b_1 is stronger than b_2 with all variables x' substituted with e. For example, $[n = n' \mid n'^2 == n]$ is refined by $n = -\sqrt{n}$ because assertion of a universally true predicate *true* always skips, so that $[n = n' \mid n'^2 == n]$ is the same as $\mathsf{assert}\ \mathit{true}; [n = n' \mid n'^2 == n]$, and also because $\mathit{true} \ \Rightarrow \ ((-\sqrt{n})^2 == n)$. Effectively, this law expresses the fact that decreasing nondeterminism is a refinement.

3.1.2 Data Refinement of Program Statements

Data refinement is a general technique by which one can change data representation in a refinement. Assume that statements S and S' operate on state spaces Σ and Σ' respectively, i.e. $S : \Sigma \mapsto \Sigma$ and $S' : \Sigma' \mapsto \Sigma'$. Let $R : \Sigma' \to \Sigma \to \mathit{Bool}$ be a relation between the state spaces Σ' and Σ. Following [4], the statement S is said to be *data refined* by the statement S' via the relation R, denoted $S \sqsubseteq_R S'$, if coercing the concrete state Σ' to the abstract state Σ followed by executing S is refined by executing S' followed by coercing the concrete state to the abstract:

$$S \sqsubseteq_R S' \quad \widehat{=} \quad \{R\}; S \sqsubseteq S'; \{R\}$$

The angelic nondeterministic assignment $\{R\}$ used here coerces the concrete state to the abstract. Usually, if the concrete state is represented by the variable $c : \Sigma'$ and the abstract one by the variable $a : \Sigma$, the relation R applied to c and a is equal to some boolean expression t which may refer to a, c and other program variables over the global state. The *abstraction statement* $\{R\}$ written in terms of program variables will then have the form $\{a = a' \mid t[a \leftarrow a']\}$, where $t[a \leftarrow a']$ is t with all occurrences of a substituted with a'.

To illustrate data refinement laws, let us present the rule for data refinement of demonic nondeterministic assignment:

$$\mathsf{assert}\ p; [a, u = a', u' \mid b_1] \ \sqsubseteq_R \ \mathsf{assert}\ p'; [c, u = c', u' \mid b_2], \tag{2}$$

$$\text{if } p \wedge t \Rightarrow p' \text{ and } p \wedge t \wedge b_2 \Rightarrow (\exists a' \bullet b_1 \wedge t')$$

Here t' is equal to t with all occurrences of a, c and u substituted with a', c' and u', i.e. $t' = t[a, c, u \leftarrow a', c', u']$. According to this rule, for example, the nondeterministic assignment to a variable e of some element of a nonempty set s is refined by the nondeterministic assignment to e of some element of a nonempty sequence l:

$$\begin{array}{l} \mathsf{assert}\ s \mathrel{!=} \emptyset; [s, e = s', e' \mid e' \in s] \sqsubseteq_R \\ \mathsf{assert}\ l \mathrel{!=} \langle\rangle; [l, e = l', e' \mid \exists i \bullet 0 \leq i < \#l \wedge e' == l[i]] \end{array}$$

Here $R\ l\ s = (\forall e \bullet e \in s == e \mathrel{\mathsf{in}} l)$ and verification of the necessary preconditions can be done using the basic properties of sets and sequences.

Presenting other rules of data refinement is outside the scope of this paper and we refer the interested reader to [5, 21] which contain large collections of refinement rules.

3.1.3 Class Refinement

Class refinement is defined to hold between classes C and D if there exists a relation R such that the constructor of C is data refined by the constructor of D with respect to R and every method of C is data refined by the corresponding method of D with respect to R. Suppose that the constructors of C and D with input parameters g_0 and g_0' of types Γ_0 and Γ_0' are specified by statements K and K' of types $\Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $\Gamma_0' \mapsto \Sigma' \times \Gamma_0'$ respectively. Further suppose that a relation $R : \Sigma \leftrightarrow \Sigma'$ coerces the attributes d of D to the attributes c of C; if D has the same attributes as C, this relation is the identity relation Id, if D inherits all the attributes of C and adds some new, this relation is the projection. Similarly, a relation $Q : \Gamma_0' \leftrightarrow \Gamma_0$ coerces the input parameter g_0' to the input parameter g_0. In case the input parameters are of the same type, Q is equal to the identity relation. Constructor refinement with respect to the relations Q and R is defined as follows:

$$\{Q\}; K \sqsubseteq K'; \{R \times \mathit{True}\}$$

Here the relational product $R \times \mathit{True}$ relates pairs of states (d, g_0') and (c, g_0) so that R holds of d and c and True holds of g_0' and g_0. As the values of the input parameters in the end of the constructors are irrelevant, coercing them using the relation True will always succeed. In terms of program variables for the attributes c of C and d of D the rule for constructor refinement can be expressed as follows:

$$\{g_0 = g' \mid Q\ g_0'\ g'\}; K \sqsubseteq K'; \{c, g_0 = c', g' \mid R\ d\ c'\}$$

Consider now refinement of statements M_i and M_i' that specify the behavior of some method called Meth_i in C and D respectively. Formally, a method with input parameters $g_i : \Gamma_i$ and an output parameter d_i of type Δ_i, operating on the attributes of type Σ is a statement of type $\Sigma \times \Gamma_i \times \Delta_i \mapsto \Sigma \times \Gamma_i \times \Delta_i$. As the types of input and return parameters of $Meth_i$ in C and D are necessarily identical, we can coerce the parameters using the identity relation. Refinement of methods with the respect to the relation R coercing the corresponding attributes is then given as follows:

$$\{R \times Id\}; M_i \sqsubseteq M_i'; \{R \times Id\}$$

In terms of program variables for the attributes c of C and d of D the rule for method refinement can be expressed as follows:

$$\{c = c' \mid R\ d\ c'\}; M_i \sqsubseteq M_i'; \{c = c' \mid R\ d\ c'\}$$

If D has new methods there is an additional proof obligation that every new method of D preserves the set of reachable states of C. This requirement is necessary because if new methods take an instance of D into a state which is perceived as unreachable in the context of C, clients of D may get invalidated. In practice, the set of reachable states is preserved by all non-modifying methods and by modifying methods that refine an arbitrary composition of the original methods. For a formal treatment of class refinement and consistency in the presence of new methods see [2].

When classes have explicit invariants, apart from proving class refinement it is necessary to verify that the classes are *consistent* with respect to the corresponding class invariants, and that these class invariants are related via an abstraction relation. Namely, if I is the invariant of C (in the case when the interclass invariant of C is different from *true*, the invariant I is the conjunction of the class invariant and the interclass invariant, otherwise it is just the class invariant), we have to prove that the constructor of C establishes I and all methods of C preserve I. Let the constructor of C be specified by a statement K, then verification of establishing I by K amounts to proving that the total correctness assertion $true \ \{\!|\ K\ |\!\}\ I$ holds. If K has some precondition p, e.g. places some restrictions on input parameters, then we have to conjoin p to the precondition of the correctness assertion, getting to prove $p\ \{\!|\ K\ |\!\}\ I$, which means that if p holds in the beginning then K guarantees to establish I in the end. Similarly, verification of preserving the invariant I by a method M_i of C requires verifying the correctness assertion $I\ \{\!|\ M_i\ |\!\}\ I$. If M itself has a precondition p_i, this correctness assertion becomes $I \wedge p_i\ \{\!|\ M_i\ |\!\}\ I$.

Coercing an abstract invariant using an abstraction relation produces an invariant on a concrete state that restricts the possible values of the concrete state as the abstract invariant restricts the possible values of the abstract state. More formally, if a and c are the program variables representing an abstract and a concrete states respectively, I and J are boolean expressions on a and c representing the corresponding invariants, and R is an abstraction relation, then I can be expressed on the concrete state c as $(\exists a \bullet R\ c\ a \wedge I)$.

In verifying class refinement between C and D, we have to prove that the invariant J of D is stronger than or equal to the invariant I of C with respect to R, i.e. $J \Rightarrow (\exists a \bullet t \wedge I)$. Verifying that this relation between the invariants holds, allows us to make sure that instances of D preserve the invariant of C with respect to the abstraction relation, which is important if they are to be dynamically substituted for instances of C. Moreover, if D is a subclass of C, self-referential method invocations in C can get redirected to D. To prevent such a *down-call* of a subclass method from a superclass method from aborting, the subclass invariant must be equal to the superclass invariant with respect to the abstraction relation, i.e. $J = (\exists a \bullet t \wedge I)$, with $=$ standing for logical equality. This condition guarantees that both C and D preserve mutual invariants, which is a critical requirement in the presence of subtyping polymorphism and possible self-referential calls between C and D. For a detailed discussion of these issues we refer to [18]. In the next subsection we will illustrate all these concepts and requirements with an example of proving class refinement between *Iterator* and *ListIterator*, both having non-trivial class invariants.

3.2 Proving Class Refinement in Practice

In proving the class refinement $\mathit{Iterator} \sqsubseteq \mathit{ListIterator}$ we have to select an abstraction relation coercing the attributes $\mathit{lst}, \mathit{ind}, \mathit{canModify}, \mathit{modified}$ of *ListIterator* to the attributes $\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{modified}, \mathit{next}$ of *Iterator*. To distinguish between the attributes *modified* in the two classes, we will call cm the one in *Iterator* and lm the one in *ListIterator*. Also, for convenience we will abbreviate $(\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{cm}, \mathit{next})$ by attr and $(\mathit{lst}, \mathit{ind}, \mathit{canModify}, \mathit{lm})$ by attr'.

The abstraction relation R can now be given as follows:

$$\begin{array}{rcl} R\ \mathit{attr}'\ \mathit{attr} & = & I' \wedge J' \wedge Q\ \mathit{lst}\ \mathit{col} \wedge \mathit{canModify} == \mathit{canRemove} \wedge \\ & & (\mathit{canModify} \Rightarrow \mathit{next} == \mathit{lst.elems}[\mathit{ind}]) \wedge \\ & & \mathit{toBag}(\mathit{lst.elems}[0..\mathit{ind}]) == \mathit{current} \end{array}$$

Here I' and J' are the combined class and interclass invariants of *Iterator* and *ListIterator* with the *modified* parameter called cm in *Iterator* and lm in *ListIterator*, and Q is an abstraction relation coercing *List* to *Collection*:

$$\begin{array}{rcl} I' & = & \mathit{col}\ != \mathit{null} \wedge \mathit{current} \subseteq \mathit{col.elems} \wedge \mathit{cm} == \mathit{col.modified} \wedge \\ & & (\mathit{canRemove} \Rightarrow \mathit{next} \in \mathit{current}) \\ J' & = & \mathit{lst}\ != \mathit{null} \wedge -1 \leq \mathit{ind} \leq \#\mathit{lst.elems} \wedge \mathit{lm} == \mathit{lst.modified} \wedge \\ & & (\mathit{canModify} \Rightarrow 0 \leq \mathit{ind} < \#\mathit{lst.elems}) \\ Q\ l\ c & = & (l == \mathit{null} \wedge c == \mathit{null}) \vee (l\ != \mathit{null} \wedge c\ != \mathit{null} \wedge \\ & & \mathit{toBag}(l.\mathit{elems}) == c.\mathit{elems} \wedge l.\mathit{modified} == c.\mathit{modified}) \end{array}$$

We distinguish the relation Q because it will be used not only as a part of R, but also as an abstraction relation coercing constructor input parameters.

3.2.1 Proving Constructor and Method Refinement

We begin with proving data refinement between the constructors of *Iterator* and *ListIterator* with respect to the relations Q and R. The goal we have to prove is as follows:

$$\begin{array}{l}
\{c = c' \mid Q\ l\ c'\};\\
\mathsf{assert}\ c \mathrel{!=} \mathit{null};\\
\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{cm}, \mathit{next} = c, [\![]\!], \mathit{false}, c.\mathit{modified}, \mathit{null}\\
\sqsubseteq\\
\mathsf{assert}\ l \mathrel{!=} \mathit{null}; \mathit{lst}, \mathit{ind}, \mathit{canModify}, \mathit{lm} = l, -1, \mathit{false}, l.\mathit{modified};\\
\{\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{cm}, \mathit{next}, c = \mathit{col}', \mathit{cur}', r', \mathit{cm}', n', c' \mid\\
\quad R\ (\mathit{lst}, \mathit{ind}, \mathit{canModify}, \mathit{lm})\ (\mathit{col}', \mathit{cur}', r', \mathit{cm}', n')\}
\end{array}$$

Deterministic assignment can always be rewritten as angelic nondeterministic assignment, according to the rule

$$x = e \quad = \quad \{x = x' \mid x' == e\} \tag{3}$$

Also, assertion can be propagated inside an adjacent angelic assignment,

$$\mathsf{assert}\ p; \{x = x' \mid b\} \quad = \quad \{x = x' \mid p \wedge b\} \tag{4}$$

$$\{x = x' \mid b\}; \mathsf{assert}\ p \quad = \quad \{x = x' \mid p[x \leftarrow x'] \wedge b\} \tag{5}$$

Applying these rules we get

$$\begin{array}{l}
\{c = c' \mid Q\ l\ c' \wedge c' \mathrel{!=} \mathit{null}\};\\
\{\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{cm}, \mathit{next} = \mathit{col}', \mathit{cur}', r', \mathit{cm}', n' \mid\\
\quad \mathit{col}' == c \wedge \mathit{cur}' == [\![]\!] \wedge r' == \mathit{false} \wedge \mathit{cm}' == c.\mathit{modified} \wedge n' == \mathit{null}\}\\
\sqsubseteq\\
\{\mathit{lst}, \mathit{ind}, \mathit{canModify}, \mathit{lm} = \mathit{lst}', \mathit{ind}', m', \mathit{lm}' \mid\\
\quad l \mathrel{!=} \mathit{null} \wedge \mathit{lst}' == l \wedge \mathit{ind}' == -1 \wedge m' == \mathit{false} \wedge \mathit{lm}' == l.\mathit{modified}\};\\
\{\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{cm}, \mathit{next}, c = \mathit{col}', \mathit{cur}', r', \mathit{cm}', n', c' \mid\\
\quad R\ (\mathit{lst}, \mathit{ind}, \mathit{canModify}, \mathit{lm})\ (\mathit{col}', \mathit{cur}', r', \mathit{cm}', n')\}
\end{array}$$

Two angelic assignment statements can be merged together according to the following rule:

$$\{x = x' \mid b\}; \{y = y' \mid c\} \quad = \quad \{x, y = x', y' \mid b \wedge c[x \leftarrow x']\} \tag{6}$$

As the abstraction statement removes concrete attributes, replacing them with abstract ones, application of the above rule gives us the following:

$$\begin{array}{l}
\{\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{cm}, \mathit{next}, c = \mathit{col}', \mathit{cur}', r', \mathit{cm}', n', c' \mid\\
\quad Q\ l\ c' \wedge c' \mathrel{!=} \mathit{null} \wedge \mathit{col}' == c' \wedge \mathit{cur}' == [\![]\!] \wedge\\
\quad r' == \mathit{false} \wedge \mathit{cm}' == c'.\mathit{modified} \wedge n' == \mathit{null}\}\\
\sqsubseteq\\
\{\mathit{col}, \mathit{current}, \mathit{canRemove}, \mathit{cm}, \mathit{next}, c = \mathit{col}', \mathit{cur}', r', \mathit{cm}', n', c' \mid\\
\quad l \mathrel{!=} \mathit{null} \wedge R\ (l, -1, \mathit{false}, l.\mathit{modified})\ (\mathit{col}', \mathit{cur}', r', \mathit{cm}', n')\}
\end{array}$$

Using the rule

$$(b \Rightarrow c) \quad \Rightarrow \quad \{x = x' \mid b\} \sqsubseteq \{x = x' \mid c\} \tag{7}$$

we can now reduce the proof to

$$\begin{array}{l}
Q\ l\ c' \wedge c' \mathrel{!}= null \wedge col' == c' \wedge cur' == \llbracket\rrbracket \wedge \\
r' == false \wedge cm' == c'.modified \wedge n' == null \\
\Rightarrow \\
l \mathrel{!}= null \wedge col' \mathrel{!}= null \wedge cur' \subseteq col'.elems \wedge cm' == col'.modified \wedge \\
(r' \Rightarrow n' \in cur') \wedge l \mathrel{!}= null \wedge -1 \le -1 \le \#l.elems \wedge \\
l.modified == l.modified \wedge (false \Rightarrow 0 \le -1 \le \#l.elems - 1) \\
Q\ l\ col' \wedge false == r' \wedge (false \Rightarrow n' == l.elems[-1]) \wedge \\
toBag(l.elems[0..-1]) == cur'
\end{array}$$

Applying simple logic transformations, we reduce this goal to true, completing our proof of constructor refinement.

For the proof of method refinement between the methods *hasNext* as defined in *Iterator* and *ListIterator*, we would need to show that the values returned in these methods are equal under the abstraction relation R:

$$R\ attr'\ attr \Rightarrow (current \subset col.elems \ = \ ind < \#lst.elems - 1)$$

We prove the boolean equality by proving mutual implications:

1. $R\ attr'\ attr \Rightarrow (current \subset col.elems \Rightarrow ind < \#lst.elems - 1)$
2. $R\ attr'\ attr \Rightarrow (ind < \#lst.elems - 1 \Rightarrow current \subset col.elems)$

For the proof of the first subgoal we use a lemma $c \subsetneq b \Rightarrow \#c < \#b$, which can easily be proved for arbitrary bags c and b, to get

$$R\ attr'\ attr \Rightarrow (\#current < \#col.elems \Rightarrow ind < \#lst.elems - 1)$$

Using the clause $toBag(lst.elems[0..ind]) == current$, which is a part of $R\ attr'\ attr$, and then a lemma $\#toBag(l) == \#l$, we get

$$\begin{array}{l}
R\ attr'\ attr \Rightarrow \\
\quad (\#(lst.elems[0..ind]) < \#col.elems \Rightarrow ind < \#lst.elems - 1)
\end{array}$$

Assuming that $ind \ge \#lst.elems - 1$ and using the definition of subsequence we get

$$\begin{array}{l}
R\ attr'\ attr \wedge ind \ge \#lst.elems - 1 \Rightarrow \\
\quad (\#lst.elems < \#col.elems \Rightarrow ind < \#lst.elems - 1)
\end{array}$$

Now from $R\ attr'\ attr$ we get that $toBag(lst.elems) == col.elems$ and, therefore, $\#lst.elems == \#col.elems$, using the abovementioned lemmas.

We reach a contradiction in the assumptions, thus proving the goal:

$$
\begin{aligned}
& R\ attr'\ attr \wedge ind \geq \#lst.elems - 1 \wedge \#lst.elems == \#col.elems \Rightarrow \\
& \quad (\#lst.elems < \#col.elems \Rightarrow ind < \#lst.elems - 1) \\
= \ & R\ attr'\ attr \wedge ind \geq \#lst.elems - 1 \wedge \#lst.elems == \#col.elems \wedge \\
& \quad \#lst.elems < \#col.elems \Rightarrow ind < \#lst.elems - 1) \\
= \ & false \Rightarrow ind < \#lst.elems - 1 \\
= \ & true
\end{aligned}
$$

The second subgoal

$$R\ attr'\ attr \Rightarrow (ind < \#lst.elems - 1 \Rightarrow current \subset col.elems)$$

is proved similarly to the first subgoal, using lemmas

$$
\begin{aligned}
& i < \#l - 1 \Rightarrow \#l[0..i] < \#l \text{ and} \\
& c \subseteq b \wedge \#c < \#b \Rightarrow c \subset b
\end{aligned}
$$

The next method refinement we must prove is between the methods *next* in *Iterator* and *ListIterator* respectively. Namely, we have to prove the following data refinement:

$$
\begin{aligned}
& \textsf{assert } current \subset col.elems; [next = e \mid e \in (col.elems \setminus current)]; \\
& current, canRemove = current + \llbracket next \rrbracket, true; \textsf{return } next \\
& \sqsubseteq_R \\
& \textsf{assert } ind < \#lst.elems - 1; ind, canModify = ind + 1, true; \\
& \textsf{return } lst.elems[ind]
\end{aligned}
$$

First of all, returning a value from a method can be modeled by assigning the returned value to a variable *res* representing the result parameter. Therefore, we can rewrite the above data refinement as follows:

$$
\begin{aligned}
& \textsf{assert } current \subset col.elems; [next = e \mid e \in (col.elems \setminus current)]; \\
& current, canRemove = current + \llbracket next \rrbracket, true; res = next \\
& \sqsubseteq_R \\
& \textsf{assert } ind < \#lst.elems - 1; ind, canModify = ind + 1, true; \\
& res = lst.elems[ind]
\end{aligned}
$$

Two demonic assignment statements can be merged together according to the following rule:

$$[x = x' \mid b]; [y = y' \mid c] \quad = \quad [x, y = x', y' \mid b \wedge c[x \leftarrow x']] \tag{8}$$

Transforming deterministic assignments into demonic assignments and applying this rule, we get

$$
\begin{array}{l}
\textsf{assert}\ \mathit{current} \subset \mathit{col.elems};\\
[\mathit{next}, \mathit{current}, \mathit{canRemove}, \mathit{res} = n', \mathit{cur}', r', \mathit{res}' \mid n' \in (\mathit{col.elems} \setminus \mathit{current}) \wedge\\
\quad \mathit{cur}' == \mathit{current} + [\![n']\!] \ \wedge\ r' == \mathit{true} \ \wedge\ \mathit{res}' == n']\\
\sqsubseteq_R\\
\textsf{assert}\ \mathit{ind} < \#\mathit{lst.elems} - 1;\\
[\mathit{ind}, \mathit{canModify}, \mathit{res} = \mathit{ind}', m', \mathit{res}' \mid\\
\quad \mathit{ind}' == \mathit{ind} + 1 \ \wedge\ m' == \mathit{true} \ \wedge\ \mathit{res}' == \mathit{lst.elems}[\mathit{ind}']]
\end{array}
$$

Applying the rule for data refinement of nondeterministic assignment statements, we can reduce the proof of this goal to two subgoals:

1. $\mathit{current} \subset \mathit{col.elems} \ \wedge\ R\ \mathit{attr}'\ \mathit{attr} \ \Rightarrow\ \mathit{ind} < \#\mathit{lst.elems} - 1$
2. $\mathit{current} \subset \mathit{col.elems} \ \wedge\ R\ \mathit{attr}'\ \mathit{attr} \wedge$
   $\mathit{ind}' == \mathit{ind} + 1 \ \wedge\ m' == \mathit{true} \ \wedge\ \mathit{res}' == \mathit{lst.elems}[\mathit{ind}']$
   $\Rightarrow$
   $(\exists \mathit{col}', \mathit{cur}', r', \mathit{cm}', n' \bullet$
   $\quad n' \in (\mathit{col.elems} \setminus \mathit{current}) \ \wedge\ \mathit{cur}' == \mathit{current} + [\![n']\!] \ \wedge\ r' == \mathit{true} \wedge$
   $\quad \mathit{res}' == n' \ \wedge\ R\ (\mathit{lst}, \mathit{ind}', m', \mathit{lm})\ (\mathit{col}', \mathit{cur}', r', \mathit{cm}', n'))$

The first subgoal, using the logical shunting rule

$$p \wedge q \Rightarrow r \ = \ p \Rightarrow (q \Rightarrow r)$$

is reduced to the first subgoal in the proof of method refinement between the methods *hasNext* and is already proved.

In the proof of the second subgoal we instantiate the existentially quantified variables by col, $\mathit{current} + [\![\mathit{lst.elems}[\mathit{ind}']]\!]$, true, cm and $\mathit{lst.elems}[\mathit{ind}']$ respectively, getting

$$
\begin{array}{l}
\mathit{current} \subset \mathit{col.elems} \ \wedge\ R\ \mathit{attr}'\ \mathit{attr} \wedge\\
\mathit{ind}' == \mathit{ind} + 1 \ \wedge\ m' == \mathit{true} \ \wedge\ \mathit{res}' == \mathit{lst.elems}[\mathit{ind}']\\
\Rightarrow\\
\mathit{lst.elems}[\mathit{ind}'] \in (\mathit{col.elems} \setminus \mathit{current}) \wedge\\
\mathit{current} + [\![\mathit{lst.elems}[\mathit{ind}']]\!] == \mathit{current} + [\![\mathit{lst.elems}[\mathit{ind}']]\!] \wedge\\
\mathit{true} == \mathit{true} \ \wedge\ \mathit{res}' == \mathit{lst.elems}[\mathit{ind}'] \ \wedge\ \mathit{col}\ != \mathit{null} \wedge\\
\mathit{current} + [\![\mathit{lst.elems}[\mathit{ind}']]\!] \subseteq \mathit{col.elems} \ \wedge\ \mathit{cm} == \mathit{col.modified} \wedge\\
(\mathit{true} \Rightarrow \mathit{lst.elems}[\mathit{ind}'] \in \mathit{current} + [\![\mathit{lst.elems}[\mathit{ind}']]\!]) \wedge\\
\mathit{lst}\ != \mathit{null} \ \wedge\ -1 \leq \mathit{ind}' \leq \#\mathit{lst.elems} \ \wedge\ \mathit{lm} == \mathit{lst.modified} \wedge\\
(m' \Rightarrow 0 \leq \mathit{ind}' < \#\mathit{lst.elems}) \ \wedge\ Q\ \mathit{lst}\ \mathit{col} \ \wedge\ m' == \mathit{true} \wedge\\
(m' \Rightarrow \mathit{lst.elems}[\mathit{ind}'] == \mathit{lst.elems}[\mathit{ind}']) \wedge\\
\mathit{toBag}(\mathit{lst.elems}[0..\mathit{ind}']) == \mathit{current} + [\![\mathit{lst.elems}[\mathit{ind}']]\!]
\end{array}
$$

Simplifying and rewriting with the definition of $R\ attr'\ attr$, we get

$$\begin{array}{l}
current \subset col.elems \ \wedge\ R\ attr'\ attr \\
\Rightarrow \\
lst.elems[ind+1] \in (col.elems \setminus current) \wedge \\
current + \llbracket lst.elems[ind+1] \rrbracket \subseteq col.elems \wedge \\
lst.elems[ind+1] \in current + \llbracket lst.elems[ind+1] \rrbracket \wedge \\
-1 \leq ind \leq \#lst.elems - 2 \wedge \\
toBag(lst.elems[0..ind+1]) == current + \llbracket lst.elems[ind+1] \rrbracket
\end{array}$$

To prove this goal, we use the following lemmas:

$$toBag(\langle e \rangle) \ = \ \llbracket e \rrbracket \tag{9}$$

$$toBag(l_1) + toBag(l_2) \ = \ toBag(l_1 \frown l_2) \tag{10}$$

$$l[0..i+1] \ = \ l[0..i] \frown \langle l[i+1] \rangle \tag{11}$$

$$b_1 \subset b_2 \wedge e \in b_2 \ \Rightarrow\ b_1 + \llbracket e \rrbracket \subseteq b_2 \tag{12}$$

$$(\exists i \mid 0 \leq i \leq \#l - 1 \bullet l[i] = e) \ \Rightarrow\ e \in toBag(l) \tag{13}$$

$$e \in b_1 \ \wedge\ b_2\ e < b_1\ e \ \Rightarrow\ e \in (b_1 \setminus b_2) \tag{14}$$

Proofs of these lemmas are straightforward from the definitions of the corresponding bag and sequence operators. Rewriting the goal with these lemmas and simplifying, we get

$$\begin{array}{l}
current \subset col.elems \ \wedge\ R\ attr'\ attr \\
\Rightarrow \\
lst.elems[ind+1] \in col.elems \ \wedge\ -1 \leq ind \leq \#lst.elems - 2 \wedge \\
current\ (lst.elems[ind+1]) < (col.elems)\ (lst.elems[ind+1]) \wedge \\
toBag(lst.elems[0..ind]) + \llbracket lst.elems[ind+1] \rrbracket == \\
\quad current + \llbracket lst.elems[ind+1] \rrbracket
\end{array}$$

Finally, using the earlier proved property

$$current \subset col.elems \ \wedge\ R\ attr'\ attr \ \Rightarrow\ ind < \#lst.elems - 1$$

and rewriting with clauses

$$\begin{array}{l}
-1 \leq ind \leq \#lst.elems \\
toBag(lst.elems[0..ind]) == current \\
toBag(lst.elems) == col.elems
\end{array}$$

from $R\ attr'\ attr$, we prove this goal.

We omit the proof of $Iterator.remove \sqsubseteq_R ListIterator.remove$ which is carried out in the same manner as the proof of $Iterator.next \sqsubseteq_R ListIterator.next$, using the same lemmas.

3.2.2 Proving Preservation of Invariants

While verifying correctness of a class having explicit invariants, we should prove that constructors of this class establish the (combined class and inter-class) invariant and all methods preserve it. Here we will only demonstrate how one can prove that methods preserve the invariant. We show that the method *add* of *ListIterator* preserves the invariant J', expressed as the following correctness assertion:

$$J' \;\{\!|\; \begin{array}{l} ind = ind + 1; [lst.elems = s \mid \exists s_1, s_2 \bullet \\ \quad s == s_1 \frown \langle o \rangle \frown s_2 \;\wedge\; lst.elems == s_1 \frown s_2 \;\wedge\; s[ind] == o]; \\ canModify, lm = false, lm + 1; lst.modified = lm \end{array} \;|\!\}\; J'$$

In the proof of this correctness assertion we will use the following rules, presented and proved in [5]:

$$p \;\{\!|\; S_1; S_2 \;|\!\}\; q \;=\; (\exists r \bullet p \;\{\!|\; S_1 \;|\!\}\; r \wedge r \;\{\!|\; S_2 \;|\!\}\; q) \tag{15}$$

$$p \;\{\!|\; x = e \;|\!\}\; q \;=\; p \subseteq q[x \leftarrow e] \tag{16}$$

$$p \;\{\!|\; x = x' \mid b \;|\!\}\; q \;=\; p \subseteq (\forall x' \bullet b \Rightarrow q[x \leftarrow x'] \tag{17}$$

Applying rules (15) and (16) and instantiating the existentially quantified predicate to J_1 such that

$$\begin{array}{rcl} J_1 & = & lst \mathrel{!=} null \;\wedge\; -1 \leq ind \leq \#lst.elems + 1 \;\wedge\; lm == lst.modified \;\wedge \\ & & (canModify \Rightarrow 0 \leq ind \leq \#lst.elems) \end{array}$$

we get to prove two subgoals

1. $J' \;\Rightarrow\; J_1[ind \leftarrow ind + 1]$
2. $J_1 \;\{\!|\; \begin{array}{l} [lst.elems = s \mid \exists s_1, s_2 \bullet \\ \quad s == s_1 \frown \langle o \rangle \frown s_2 \;\wedge\; lst.elems == s_1 \frown s_2 \;\wedge\; s[ind] == o]; \\ canModify, lm = false, lm + 1; lst.modified = lm \end{array} \;|\!\}\; J'$

The first subgoal obviously holds, since $-1 \leq ind \leq \#lst.elems \;\Rightarrow\; -1 \leq ind + 1 \leq \#lst.elems + 1$ and $0 \leq ind < \#lst.elems \;\Rightarrow\; 0 \leq ind + 1 \leq \#lst.elems$. For proving the second subgoal we apply rules (15) and (17), instantiating the existentially quantified predicate to J':

1. $lst \mathrel{!=} null \;\wedge\; -1 \leq ind \leq \#lst.elems + 1 \;\wedge\; lm == lst.modified \;\wedge$
   $(canModify \Rightarrow 0 \leq ind \leq \#lst.elems)$
   $\Rightarrow$
   $(\forall s \bullet (\exists s_1, s_2 \bullet s == s_1 \frown \langle o \rangle \frown s_2 \;\wedge\; lst.elems == s_1 \frown s_2 \;\wedge\; s[ind] == o)$
   $\quad \Rightarrow lst \mathrel{!=} null \;\wedge\; -1 \leq ind \leq \#s \;\wedge\; lm == lst.modified \;\wedge$
   $\quad (canModify \Rightarrow 0 \leq ind \leq \#s - 1))$
2. $J' \;\{\!|\; canModify, lm = false, lm + 1; lst.modified = lm \;|\!\}\; J'$

Using simple logic transformations, the first of these goals can be reduced to

$$\begin{array}{l} -1 \le ind \le \#lst.elems + 1 \;\wedge\; (canModify \Rightarrow 0 \le ind \le \#lst.elems) \wedge \\ s == s_1 \frown \langle o \rangle \frown s_2 \;\wedge\; lst.elems == s_1 \frown s_2 \\ \Rightarrow \\ -1 \le ind \le \#s \;\wedge\; (canModify \Rightarrow 0 \le ind \le \#s - 1) \end{array}$$

and proved using the lemma $\#(l_1 \frown l_2) \;=\; \#l_1 + \#l_2$.

For the proof of the second subgoal, we apply rules (15) and (16) instantiating the existentially quantified predicate to J_2 such that

$$\begin{array}{lcl} J_2 & = & lst \mathrel{!=} null \;\wedge\; -1 \le ind \le \#lst.elems \;\wedge\; lm == lst.modified + 1 \wedge \\ & & (canModify \Rightarrow 0 \le ind < \#lst.elems) \end{array}$$

The resulting subgoals

1. $lst \mathrel{!=} null \;\wedge\; -1 \le ind \le \#lst.elems \;\wedge\; lm == lst.modified \wedge$
   $(canModify \Rightarrow 0 \le ind < \#lst.elems)$
   $\Rightarrow$
   $lst \mathrel{!=} null \;\wedge\; -1 \le ind \le \#lst.elems \;\wedge\; lm + 1 == lst.modified + 1 \wedge$
   $(false \Rightarrow 0 \le ind < \#lst.elems)$
2. $lst \mathrel{!=} null \;\wedge\; -1 \le ind \le \#lst.elems \;\wedge\; lm == lst.modified + 1 \wedge$
   $(canModify \Rightarrow 0 \le ind < \#lst.elems)$
   $\Rightarrow$
   $lst \mathrel{!=} null \;\wedge\; -1 \le ind \le \#lst.elems \;\wedge\; lm == lm \wedge$
   $(canModify \Rightarrow 0 \le ind < \#lst.elems)$

hold trivially.

This completes our proof of method *ListIterator.add* preserving the combined class and interclass invariant of *ListIterator*. Proofs of invariant preserving for other methods of *ListIterator* can be carried out in a similar manner. Naturally, the same principles apply to proving consistency of methods of *Iterator* with respect to its combined invariant. Moreover, non-modifying methods, such as *hasNext*, *hasPrevious*, *nextIndex*, and *previousIndex*, preserve the corresponding invariants automatically.

When proving refinement between two classes having explicit invariants, we should also verify that the concrete invariant is stronger than or equal to the abstract invariant with respect to an abstraction relation. As in our case *Iterator* and *ListIterator* are the specifications of the corresponding interfaces and *ListIterator* does not inherit from *Iterator*, a stronger requirement that the invariants I' and J' must be equal does not apply. Therefore, we have only to prove that J' is stronger than or equal to I' with respect to R, i.e.

$$J' \;\Rightarrow\; (\exists attr \bullet R\ attr'\ attr \;\wedge\; I')$$

where $attr$ abbreviates $(col, current, canRemove, cm, next)$ and $attr'$ abbreviates $(lst, ind, canModify, lm)$, and also I' and J' are the combined invariants of *Iterator* and *ListIterator* with the *modified* parameter called cm in *Iterator* and lm in *ListIterator*. Rewriting with the definitions of I' and J', we get to prove the following goal:

$$
\begin{array}{l}
lst \mathrel{!=} null \wedge -1 \leq ind \leq \#lst.elems \wedge lm == lst.modified \wedge \\
(canModify \Rightarrow 0 \leq ind < \#lst.elems) \\
\Rightarrow \\
(\exists attr \bullet R\ attr'\ attr \wedge col \mathrel{!=} null \wedge \\
current \subseteq col.elems \wedge cm == col.modified \wedge \\
(canRemove \Rightarrow next \in current))
\end{array}
$$

Instantiating the existentially quantified variables $attr$ so that col is instantiated with a reference c to an object with $elems == toBag(lst.elems)$ and $modified == lst.modified$, $canRemove$ is instantiated with $canModify$, $current$ with $toBag(lst.elems[0..ind])$, cm with $c.modified$, and $next$ with $lst.elems[ind]$, we can prove this goal as well.

4 Conclusions and Related Work

In this paper we present a novel approach to specification and verification of object-oriented frameworks. The novelty of our approach is in blurring out the difference between specifications and implementations which permits abstracting away from implementation details in a specification, yet allowing to be precise about important behavioral issues, such as, e.g., a fixed method invocation order or an iterative execution of a particular statement. The benefits of combining executable statements with specification statements when reasoning about object-oriented and component-based systems are described by Martin Büchi and Emil Sekerinski in [7]. In particular, they note that a popular form of specification in terms of pre- and post-conditions does not scale well to reasoning about object-oriented and component-based systems, because pre- and post-conditions, being predicates, cannot contain calls to other methods, except when the latter are pure functions. Therefore, one has to reinvent the wheel every time when specifying the behavior of a method implementing some functionality by calling other methods. Specifications in terms of abstract statements, as pointed out in [7], are not affected by this scalability problem. Also, Büchi and Sekerinski note that pre- and post-conditions, which are only checked at runtime, help to locate errors but do not prevent them as does static analysis.

Our specification methodology is supported by a solid formal foundation: every executable statement of the Java language as well as every specification statement that we use has a precise mathematical meaning as described in [19, 3]. Moreover, treating specifications and implementations in a uniform logical framework permits formal reasoning about their relationship and properties. Of particular interest is the verification of behavioral conformance between specifications of interfaces and implementations of these interfaces. Verifying behavioral conformance of implementations to their specifications as well as behavioral conformance of subinterface specifications to the corresponding superinterface specifications permits ensuring correctness of the whole system.

We illustrate our specification method by specifying a part of Java Collections Framework. We have developed formal specifications of other subinterfaces of *Collection* as well, but omit them for the reasons of limited space. In the process of specifying JCF interfaces we have identified a number of ambiguities and inconsistencies in the informal description of the behavior associated with the interfaces. Our specification, being both succinct and precise, clarifies the found ambiguities and eliminates the inconsistencies. The difference in size between the implementation of *Collection*'s *contains* method as given in the class *AbstractCollection*, which is a part of the standard JCF implementation, and between our specification of this method is quite illustrative of the general picture. In *AbstractCollection* the method *contains* is defined by

```
public boolean contains(Object o) {
    Iterator e = iterator();
        if (o==null) {
            while (e.hasNext())
                if (e.next()==null) return true;
        } else {
            while (e.hasNext())
                if (o.equals(e.next())) return true;
        }
    return false;
}
```

while in our specification it is defined by return $(o \in elems)$.

Related work in formal specification of object-oriented systems includes, but is not limited to, [9, 11]. William Cook in [9] specifies Smalltalk-80 collection class library. Although the library is organized by inheritance, Cook argues that interface inheritance or subtyping is a logical basis for the library organization, supporting this claim by specifying the interfaces and revealing several problems with the current organization of the library. With the Java Collection Library that we specify here, interface inheritance is separated from the implementation inheritance and, since the former forms the basis for polymorphic object substitutability in client programs, we associate behavioral specifications with interfaces, as does Cook. One of the main differences of our work from that of Cook is that his specifications are given in terms of pre- and post-conditions, following Pierre America's approach in [1], while we use a specification language combining specification statements with executable ones. More importantly, America's approach used by Cook is rigorous rather than formal and the specification language does not have a formal semantics. In our language, as was explained above, every statement has a precise mathematical meaning, and reasoning about our specifications and their relation to executable Java programs can be carried out completely formally, in a unified logical framework.

David Egle in [11] specifies the Microsoft Foundation Class Library to evaluate Larch/C++ as a specification language. The specification language, having no formal semantics, includes constructs like "ensures informally". The specification is based on pre- and post-conditions (called "requires" and "ensures", respectively) and suffers from the non-scalability problem discussed above. The author concludes the evaluation by saying that it is his belief that "formal specification using Larch/C++ is good for specifying things more precisely and unambiguously, but is too rigorous in some respects". We, on the other hand, believe that our specification of Java Collections Framework is both precise and general enough to permit different implementations: the possibility to include method calls in the specification eliminates the need for approximating the behavior of these methods in pre- and post-conditions, whereas the availability of nondeterministic statements allow us to abstract away from unnecessary implementation details and express the behavior common to several implementations.

The detailed elaboration of our formalization of object-oriented constructs and mechanisms, as described in [19, 3], opens the possibility of mechanized reasoning and mechanical verification. An interesting recent work by Bart Jacobs et al. in [15] reports a work in progress on building a front-end tool for translating Java classes to higher-order logic in PVS [22]. The authors state that "current work involves incorporation of Hoare logic [13], via appropriate definitions and rules in PVS", and present in [15] a description of the tool "directly based on definitions". In this work we test the applicability of the theoretic foundation for reasoning about object-oriented programs developed in [19, 3]. This theoretic foundation is based on the logical framework for reasoning about imperative programs. A tool supporting verification of correctness and refinement of imperative programs and known as the Refinement Calculator [16] already exists and extending it to handling object-oriented programs, including Java programs, appears to be only natural.

There are a few issues that we haven't addressed in this project, in particular, the role of exceptions, their relation to assertion statements and their formal semantics are left as a topic for future work. Method early returns are treated somewhat informally: we assume that every method returning the result inside the conditional statement or inside the loop can be rewritten to an equivalent one returning the result as the last operation. Formal treatment of early returns represents an interesting research topic.

Acknowledgement

We would like to thank Joakim von Wright for useful comments on this paper.

References

[1] P. America. Designing an object-oriented programming language with behavioral subtyping. In J. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, LNCS 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.

[2] R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct object substitutability. http://www.abo.fi/~amikhajl/Papers/ClassRefSem.ps.

[3] R. Back, A. Mikhajlova, and J. von Wright. Class refinement as semantics of correct subclassing. Technical Report 147, Turku Centre for Computer Science, December 1997.

[4] R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences*. IEEE, January 1989.

[5] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, April 1998.

[6] J. Bloch. Java Collections Framework: Collections 1.2. http://java.sun.com/docs/books/tutorial/collections/index.html.

[7] M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In *Second Workshop on Component-Oriented Programming (WCOP'97) held in conjunction with ECOOP'97*, June 1997.

[8] M. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, and J. von Wright. The refinement calculator: Proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific'97: Proceedings of FMP'97*, Discrete Mathematics & Theoretical Computer Science, pages 40–61, Wellington, New Zealand, July 1997. Springer-Verlag.

[9] W. R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA'92*, pages 1–15. ACM SIGPLAN Notices, 27(10), October 1992.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.

[11] D. M. Egle. Evaluating Larch/C++ as a specification language: A case study using the Microsoft Foundation Class Library. Technical Report TR95-17, Iowa State University, July 1995.

[12] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, LNCS 213. Springer-Verlag, 1986.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–583, 1969.

[14] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[15] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *Proceedings of OOPSLA'98*, pages 329–340, Vancouver, Canada, Oct. 1998. Association for Computing Machinery.

[16] T. Långbacka, R. Ruksenas, and J. von Wright. TkWinHOL: A tool for window inference in HOL. *Higher Order Logic Theorem Proving and its Applications: 8th International Workshop*, 971:245–260, September 1995.

[17] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[18] A. Mikhajlova. Consistent extension of components in presence of explicit invariants. In W. Weck, J. Bosch, and C. Szyperski, editors, *Third International Workshop on Component-Oriented Programming (WCOP'98) held in conjunction with ECOOP'98*. TUCS General Publication Series, No. 10, July 1998.

[19] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of 4th International Formal Methods Europe Symposium, FME'97*, LNCS 1313, pages 82–101. Springer, 1997.

[20] C. C. Morgan. Data refinement by miracles. *Information Processing Letters*, 26:243–246, January 1988.

[21] C. C. Morgan. *Programming from Specifications*. Prentice–Hall, 1990.

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**