The CleanJava Language for Functional Program Verification

Yoonsik Cheon, Cesar Yeep, and Melisa Vela

TR #11-52
August 2011; revised December 2011

Keywords: formal specification, functional program verification, intended function, CleanJava language.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification—correctness proofs, formal methods; D.3.2 [*Programming Languages*] Language Constructs and Features—classes and objects, control structures, inheritance; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—assertions, logics of programs, specification techniques.

This is an extended version of a paper appeared in *ITNG 2011: 8th International Conference on Information Technology, April 11-13, 20119, Las Vegas, NV*, pages 221-226, IEEE Computer Society. To appear in a special issue of the *International Journal of Software Engineering.*

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

The CleanJava Language for Functional Program Verification

Yoonsik Cheon(1), Cesar Yeep(1), and Melisa Vela(1)

(1) Department of Computer Science, University of Texas at El Paso, El Paso, Texas, U.S.A.
E-mail: ycheon@utep.edu, {ceyeep, smvelaloya}@miners.utep.edu

ABSTRACT

Unlike Hoare-style program verification, functional program verification supports forward reasoning by viewing a program as a mathematical function from one program state to another and proving its correctness by essentially comparing two mathematical functions, the function computed by the program and its specification. Since it requires a minimal mathematical background and reflects the way that programmers reason about the correctness of a program informally, it can be taught and practiced effectively. However, there is no formal notation supporting the functional program verification. In this article, we describe a formal notation for writing functional program specifications for Java programs. The notation, called *CleanJava*, is based on the Java expression syntax and is extended with a mathematical toolkit consisting of sets and sequences. The vocabulary of CleanJava can also be enriched by introducing user-specified definitions such as user-defined mathematical functions and specification-only methods. We believe that CleanJava is a good notation for writing functional specifications and expect it to promote the use of functional program verifications by being able to specify a wide range of Java programs.

Keywords: CleanJava, formal specification, functional program verification, intended function

1- INTRODUCTION

In the late 70s, Harlan Mills and his colleagues at IBM developed an approach to software development called *Cleanroom Software Engineering* [23] [26] [27]. Its name was taken from the electronics industry, where a physical clean room exists to prevent introduction of defects during hardware fabrication, and the method reflects the same emphasis on defect prevention rather than defect removal. Special methods are used at each stage of the software development—from requirement specification and design to implementation—to avoid errors. In particular, it uses specification and verification, where verification means proving mathematically that a program agrees with its specification.

Cleanroom is a lightweight, or semi-formal, method and tries to verify the correctness of a program using a technique that we call *functional program verification* [4] [31]. The technique requires a minimal mathematical background by viewing a program as a mathematical function from one program state to another and by using equational reasoning based on sets and functions. In essence, the functional verification involves (a) calculating the function computed by code called a *code function* and (b) comparing it with the intention of the code written as a function called an *intended function* [31]. For this, the behavior of each section of code is documented, as well as the behavior of the whole program. The documented behavior is the specification to which the correctness of a program is verified.

We believe that the functional program verification technique can be effectively taught and practiced, as it requires a minimal mathematical background and reflects the way that programmers reason about the correctness of a program informally by supporting forward reasoning [7] [32]. It is also our conjecture that if programmers become proficient in the functional program verification, they may be able to learn easily other verification techniques such as Hoare logic [10] as complementary reasoning techniques.

However, there is no formal notation or language to support the functional program verification. This not only limits the adoption of functional verification both in academia and industry but also makes it difficult to develop a standard set of support tools, thus limiting its user base [6] [7] [29] [32].

In this article we introduce a formal annotation language for the Java programming language to support Cleanroom-style functional program verification. Our language, called *CleanJava*, is based on the Java expression syntax extended with a mathematical toolkit including sets and sequences. Some notable features of CleanJava include: (a) use of Java expression syntax, (b) extensible vocabularies through user-defined functions and specification-only methods, (c) a tunable, wide-spectrum of formality, (d) support for abstraction and modularity, and (e) support for object-oriented concepts such as specification inheritance. We believe that CleanJava is a good notation for writing intended functions and will facilitate formal correctness verification and reasoning of Java programs. We are still evaluating the CleanJava language through case studies and teaching it along with functional program verification in several undergraduate and graduate-level programming and software engineering courses. However, we believe that CleanJava is a good notation for writing intended functions and will facilitate formal correctness verification and reasoning of Java programs, and this article provides a tutorial introduction to the CleanJava language describing its main features and some of the interesting design decisions as well.

The rest of this article is organized as follows. Section 2 below provides a brief overview of the functional program verification using a small running example. Section 3 is the main body of this article and describes the key features of the CleanJava language, including (a) its expression syntax for writing intended functions, (b) mechanisms for extending vocabularies for writing intended functions, such as user-defined mathematical functions and specification-only methods, (c) an approach for writing abstract specifications to support a modular specification and verification, and (d) inheritance of specifications. Section 4 describes addition features of CleanJava, such as object equality, splitting and composing specifications, and the built-in standard library. Section 5 discusses some of the most-closely related work, and Section 6 concludes this article.

2- FUNCTIONAL PROGRAM VERIFICATION

2-1 Programs as Functions

An execution of a program produces a side-effect on a program state by changing the values of some state variables such as program variables. In the functional program verification, a program execution is modeled as a mathematical function from one program state to another, where a program state is a mapping from state variables to their values. For example, consider the following code that swaps the values of two variables *x* and *y*.

```
x = x + y;
y = x - y;
x = x - y;
```

Its execution can be modeled as a mathematical function that, given a program state, produces a new state in which *x* and *y* are mapped to the initial values of *y* and *x*, respectively. The rest of the state variables, if any, are mapped to their initial values; that is, their values remain the same.

A succinct notation, called a *concurrent assignment*, is used to express these functions by only stating changes in an input state [1] [31]. A concurrent assignment is written as $[x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$ and states that each x_i's new value is e_i, evaluated concurrently in the initial state, i.e., the input state or the state just before executing the code. The value of a state variable that does not appear in the left-hand side of a concurrent assignment remains the same. For example, the function that swaps two variables *x* and *y*, is written as $[x, y := y, x]$. The concurrent assignment notation can be used to express both the actual function computed by a section of code, called a *code function*, and our intention for the code, called an *intended function* [4] [31].

2-2 Correctness Verification

The correctness of code can be verified by comparing its code function to its intended function. A program or a section of code with an intended function *f* is correct if it has a code function *p* that satisfies the following two conditions.

- The domain of *p* is a superset of the domain of *f*, i.e., $\mathrm{dom}(p) \supseteq \mathrm{dom}(f)$.
- For every *x* in the domain of *f*, *p* maps *x* to the same value that *f* maps to, i.e., $p(x) = f(x)$ for $x \in \mathrm{dom}(f)$.

We also say that p is a *refinement* of f, denoted by $p \sqsubseteq f$ [1] [31]. Since the domains of both code and intended functions are program states, the first condition means that an implementation can allow more input states than what its specification allows; that is, an implementation can do more than what its specification says, but not less.

For a correctness verification of code, we write an intended function for each section of the code. For example, the following code annotated with intended functions finds the largest element contained in a non-empty array *a*.

```
// f0: [r := largest value in a]
  // f1: [r, i := a[0], 1]
    r = a[0];
    int i = 1;

  // f2: [r, i := max of r and the largest value contained in a[i..], ?]
    while (i < a.length) {
      // f3: [r, i := max of r and a[i], i+1]
        if (a[i] > r) {
          r = a[i];
        }
        i++;
    }
```

An indentation is used to indicate the region of code that an intended function annotates. For example, the intended function f_0 documents the behavior of the whole code, and f_1 and f_2 document the behavior of the initialization code and the loop, respectively. In function f_2, a question mark (?) is used to indicate that we don't care about the final value of a loop variable *i*. The verification of the above code requires discharging the following four proof obligations.

1) $f_1; f_2 \sqsubseteq f_0$, i.e., proof that f_1 followed by f_2 is a refinement of f_0. The notation $f_1; f_2$ means a sequential composition of two functions f_1 and f_2.
2) Refinement of f_1, i.e., correctness of f_1's code.
3) Refinement of f_2, which requires the following three sub-proofs.
    a) Termination of the loop
    b) Basis step: $\neg(i < a.\text{length}) \Rightarrow I \sqsubseteq f_2$, where I denotes an identity function.
    c) Induction step: $i < a.\text{length} \Rightarrow f_3; f_2 \sqsubseteq f_2$
4) Refinement of f_3, correctness of f_3's code.

As an instructive example verification, we show below a proof of the first obligation, $f_1; f_2 \sqsubseteq f_0$.

$f_1; f_2 \equiv [r, i := a[0], 1];$
$\quad [r, i := \textit{max of r and the largest value contained in a[i..]}, ?]$

≡ [*r, i* := *max of a[0] and the largest value contained in a[1..]*, ?]
≡ [*r, i* := the *largest value in a*, ?]
⊑ [*r* := *largest value in a*]
≡ f_0

In a functional verification, the proof is sometimes straightforward because one can calculate code functions and compare them with intended functions. However, one often need to use different techniques such as a case analysis and an induction based on the structure of the code as in the proof of f_2 above [4] [31].

In the example above, we used informal English texts to describe and manipulate intended functions. In the following section, we show how to formalize them in CleanJava.

3- THE CLEANJAVA LANGUAGE

3-1 The Core Notation

CleanJava is a formal notation for annotating Java code with intended functions. It supports a rigorous and formal verification of Java code. Here we first describe the core part of the CleanJava language focusing on its syntax for writing expressions, intended functions, and annotations.

In CleanJava, an intended function is written using an extended form of Java expressions. However, CleanJava expressions have a restriction in that they cannot have side effects. Thus, Java's assignment expressions (=, +=, etc.) and increment (++) and decrement (–) operators are not allowed in CleanJava expressions, and only query methods are allowed. A query method is a method that doesn't have a side effect; it is used to ask about the state of an object without changing it. Below is the sample code of the previous section annotated with intended function written in CleanJava.

```
//@ [r := a->iterate(int x, int m = a[0] | x > m ? x : m)]
 //@ f1:[r, i := a[0], 1]
  r = a[0];
  int i = 1;

 /*@ f2: [r, i := Math.max(r, (* the largest value contained in a[i..] *)),
  @            anything] @*/
  while (i < a.length) {
   //@ [r, i := Math.max(r, a[i]), i+1]
    if (a[i] > r) {
     r = a[i];
    }
    i++;
  }
```

As shown, a CleanJava annotation is written in a special kind of comments either preceded by //@ or enclosed in a pair of /*@ and @*/ symbols. The first form is for writing a single line annotation, and the second is for writing a multi-line annotation. In an annotation line, the initial white-space and any immediately following at-signs (@) character are ignored. As before, an indentation is used to denote the section of code that an intended function annotates. The first annotation, for example, describes the behavior of the whole code, and the second describes that of the initialization code. An intended function can have an optional label or name such as f_1 and f_2.

The first annotation shows an example of CleanJava extensions to the Java expression syntax. The *iterate* operation is one of the several CleanJava-specific operations defined on arrays and collections. It has a general form of *iterate*(T_1 *x*, T_2 *y* |*E*(*x*)), where T_1 is the element type of an array or collection and *E*(*x*) is an expression of T_2 written in terms of *x*. The variable *x* is an iterator that bounds to each value of the array or collection, and *y* is an accumulator that contains the value of *E*(*x*) after each evaluation of it. The operation evaluates *E*(*x*) for each element in the array or collection, bound to *x*, storing the result of each evaluation to *y*, and returns the final value stored in *y*. The above *iterate* operation returns the largest value contained in the array *a*. Note that an arrow notation (->) is used to indicate an invocation of an iteration operation. There are several other iteration operations defined on arrays and collections, including *select*, *reject*, *collect*, *forAll*, and *exists* (see Table 1 below).

Table 1: CleanJava iteration operations defined on arrays and collections

Operator	**Description**
any(*E*)	Any element for which *E* is true
collect(*E*)	A collection that results from evaluating *E* for each element
exists(*E*)	Has at least one element for which *E* is true?
forAll(*E*)	Is *E* true for all elements?
isUnique(*E*)	Does *E* have unique values for all elements?
iterate(T_1 x_1, T_2 x_2; *E*)	Iterates over all elements accumulating the result to x_2
one(*E*)	Has only one element for which *E* is true?
reject(*E*)	A collection containing all elements for which *E* is false
select(*E*)	A collection containing all elements for which *E* is true

The annotation defining f_2 shows several features of CleanJava. First, a Java method such as *Math.max* can be used in CleanJava expressions as long as it has no side effect. Second, the keyword *anything* indicates that one doesn't care about the final value of a variable—a local or incidental variable. It is one way to write a loose specification since an arbitrary value can be assigned to such a variable by an implementation. Lastly, when writing an intended function, one can escape from formality by using an *informal description*. An informal description of the form (* *some text* *) is convenient when the formal statement is not easier to write down or clearer. It allows informal texts to be combined with formal statements and is convenient for organizing an informal documentation. Informal specifications can also be very useful when there's not enough time to develop a formal description of some aspect of the code. This kind of escape from formality is very useful, in general, to avoid describing the entire world formally when writing a specification of some code. However, there are several drawbacks to using informal descriptions. A major drawback is that informal descriptions are often ambiguous or incomplete. Another problem is that informal descriptions cannot be manipulated by tools.

3-2 Extension Mechanisms

One feature of CleanJava is that its vocabularies are not limited to a predefined set of symbols and expressions but can be extended by a programmer. In this section we describe two such extension mechanisms: user-defined functions and model methods.

In CleanJava, a programmer can introduce new mathematical functions for use in writing intended functions. For example, the following code is from the previous section with its annotations rewritten using a user-defined function.

```
//@ fun max(a) = a->iterate(int e, int m=0 | e > m ? e : m)
//@ [r := max(a)]
  //@ [r, i := a[0], 1]
    r = a[0];
    int i = 1;

  /*@ [r, i := Math.max(r, m), anything] where
     @  int m = max(Arrays.copyOfRange(a,i,a.length)) @*/
    while (i < a.length) {
      //@ [r, i := Math.max(r, a[i]), i+1]
        if (a[i] > r) {
          r = a[i];
        }
        i++;
    }
```

The first annotation introduces a function named *max* that takes an array or collection of integers and returns a maximum value contained. The body of the function is just a Java expression with CleanJava extensions such as collection operations. As shown, one doesn't have to specify the signature—argument and return types—of a function. As in modern functiona progrannung languages such as SML [24] and Haskell [12], they are automatically inferred at compile time. A CleanJava function follows the Java scoping rules [9]. Thus, the function *max* can be used in the specifications of the top-level intended function at line 2 and that of the *while* statement. It is also possible to introduce a function as a member of a class or an interface (see an example below). The fourth annotation, the intended function for the *while* statement, introduces a constant function named *m* written in terms of the user-defined function *max*. It is a local function indicated by the keyword *where*; it is visible only in the preceding intended function. A *where* clause introduces local names such as *m* that can span over multiple expressions, e.g., a whole annotation.

The scope of a user-defined function at a statement level is similar to that of a Java local variable declared in the body of a method or a constructor (refer to [9] for Java's scope rules). For example, a functions defined in a block is only accessible from within the block. The scope of the function is the block in which it is defined. A nested block can also access a function defined in the outer block. However, it cannot define a function with the same name as the one defined in the outer block. If a method declares a local function that has the same name as a class-level function (see below), the former will shadow the latter. To access the class-level function from inside the method body, use the *this* keyword as in Java.

A user-defined function can also be declared at a class level, and such a function is called a *user-defined member function*. A member function is useful when more than one method or a client of a class need to use a user-defined function. For example, the following code introduces a user-defined member function named *sumPos* and uses it to document the behavior of two different methods.

```
public class ArrayUtil {
  //@ public static fun sumPos(a) = a->iterate(int e; int m=0 | e > m ? e : m)

  //@ [a != null -> result : = sumPos(a)]
  public static int sumOfPositives(int[] a) { ... }

  /*@ [a != null && hasPos(a) -> result := sumPos(a) / cntPos(a)] where
    @   boolean hasPos(a) = a->exists(int e; e > 0),
    @   int cntPos(a) = a->select(int e; e > 0).size() @*/
  public static int averageOfPositives(int [] a) { ... }
}
```

The above code also shows the use of an intended function to define the behavior of a method for clients, i.e., as an API specification of a method. In such a use, a keyword *result* denotes the return value of a method. The *sumOfPositives* method is partial in that its behavior is defined only when the given argument *a* is not null. The optional condition preceding the arrow symbol (->) specifies the domain of the intended function[1]; if omitted, the intended function is a total function.The third annotation also illustrates how to use a *where* clause to introduce local functions. As shown in the above example, the syntax of declaring a member function is the same as that of a statement-level function except that (a) it is declared at a class level inside an annotation, and (b) it can have optional Java modifiers such as *public*, *protected*, *private*, *static*, and *stricfp*. The meanings of these modifiers are exactly the same as those in Java [9]. For example, a public member function can be accessed by the public clients of a class and is inherited by a subclass, and a static member function cannot refer to non-static class members such as non-static fields and methods.

In addition to user-defined functions, one can also introduce Java methods specifically for writing intended functions. These specification-only methods are called *model methods* [5] [16]. Figure 1 shows an example use of a model method. The class AddressBook stores entries called *contacts*; each contact consists of a few standard fields such as name, address, telephone number, and e-mail address. It defines several public methods for manipulating the contained contacts.

The specification of the *addContact* method is interesting. It is written in terms of the *append* method of which definition appears inside an annotation. The fact that the definition of the *append* method is an annotation indicates that it is a model method, meaning that it can be used only in annotations but not in Java code. The *append* method returns an array that contains the contents of the field *contacts* with the argument *c* appended; it may creates a new array to append the given contact. A model method such as the *append* method should not have a side-effect because it will be used in annotations. Except for this, its use is the same as that of a Java method. It follows Java's visibility and scoping rules [9]. The *append* method, for example, can be used in the annotations of the client code of the AddressBook class and is inherited to subclasses because it is a public method. The specification of the *hasContact* method is also interesting. It refers to a user-defined function, *has*, which is declared to be a member function. Like a model method, a member function such as the *has* function also follows Java's visibility and scoping rules [9]; it can be used in the client annotations and is inherited to subclasses.

Both user-defined functions and model methods allow one to extend the vocabularies of CleanJava. If the result or return value can be expressed in a single expression, a user-defined function would be a better choice since it provides a succinct notation. On the other hand, if it can be better expressed algorithmically as a sequence of statements, a model method would be a better choice.

[1] This extended form of a concurrent assignment is called a *conditional concurrent assignmetn*.

```
class AddressBook {

  private Contact[] contacts;

  private int size;

  //@ [contacts, size := new Contact[100], 0]
  public AddressBook() {
    contacts = new Contact[100];
    size = 0;
  }

  /@ [!hasContact(n) ->  contacts, size := append(new Contact(n,i)), size+1]
  public void addContact(String n, ContactInfo i) { ... }

  /*@ public Contact[] append(Contact c) {
    @   Contact[] cs = contacts;
    @   if (size > contacts.length - 1) {
    @     cs = new Contact[contacts.length * 2];
    @     System.arraycopy(contacts, 0, cs, 0, contacts.length);
    @   }
    @   cs[size] = c;
    @   return cs;
    @*/

  //@ [result := has(Arrays.copyOf(contacts,size), n)]
  public boolean hasContact(String n) { ... }

  //@ public fun has(a,n) = a->exists(Contact c| c.getName().equals(n))
}
```

Figure 1: An example annotation written using a model method

3-3 Support for Abstraction

CleanJava provides several features to support a modular specification and verification. A verification of client code of a class is said to be *modular* if a change on the hidden implementation details of the class such as data structures and algorithms doesn't require a re-verification of the client code [4] [14]. For a modular verification, the specification and verification of client code of a class shouldn't rely on the implementation details of the class. This in turn means that the specification of the class itself shouldn't refer to, or expose, the hidden implementation details and decisions because it is this specification that is used in the specification and verification of the client code. Otherwise, the client code is tightly coupled to the class by exploiting an exposed implementation detail or decision of the class. Its verification or reasoning will not be modular because a change on the class requires a re-specification of the class, which in turn requires a respecification and re-verification of the client code itself. In short, a class specification—as a formal API document—should be *abstract* and support information hiding in that it shouldn't refer to, or expose, hidden implementation details or decisions.

In CleanJava, one can write an abstract specification for a class that doesn't expose implementation details of the class. This is done by writing a specification that manipulates abstract values of a class, not its concrete representation values [11]. For example, in the previous section, an address book is implemented as an array of contacts, and its specification is written in terms of this array, thus exposing the hidden representation. However, for a specification purpose, an address book can be viewed, modeled, and manipulated as a set of contacts. In CleanJava, this can be achieved by using a specification-only variable, called a *model variable* [3] [5], which is similar to a model method introduced in the previous section.

Figure 2 shows an abstract specification of the AddressBook class written using a model variable. The first annotation introduces a model variable named *cset*. A generic class CJSet is a standard library class of CleanJava and provides an abstraction of a mathematical set, similar to that of java.util.Set (see Section 4-3). However, one key difference is that it is an immutable type to be used in CleanJava annotations, and thus there is no method defined that has a side-effect. The *add* method, for example, returns a new set instead of mutating the receiver. Since a model variable such as *cset* is used only in annotations, its value is not directly assigned but is given implicitly as a mapping from program variables. This mapping is called an *abstraction function* [11] [18] and is specified by an optional initializer of a model variable. For example, the value of variable *cset* is *toSet(contacts,size)*, where *toSet* is a user-defined function.

Once the abstract values of a class are defined using model variables, they can be used to write specifications for public methods of the class. For example, the intended functions of the constructor and methods such as *hasContact*, *addContact*, and *getContact* of AddressBook are written by referring to the model variable *cset*. One can also write multiple specifications for the same method, for example, a public specification written in terms of abstract values and a private specification written in terms of concrete representation values (see below).

```
//@ [cset := new CJSet<Contact>()]
public AddressBook() {
  //@ [contacts, size := new Contact[100], 0]
    contacts = new Contact[100];
    size = 0;
}
```

```
class AddressBook {

  private Contact[] contacts;

  private int size;

  /*@ public CJSet<Contact> cset = toSet(contacts,size) where
    @   fun toSet(a,0) = new CJSet<Contact>()
    @   fun toSet(a,i) = toSet(a,i-1).add(a[i]); @*/

  //@ [cset := new CJSet<Contact>()]
  public AddressBook() { ... }

  //@ [result := cset->exists(c: Contact | cgetName().equals(n))]
  public boolean hasContact(String n) { ... }

  //@ [!hasContact(n) -> cset := cset.add(new Contact(n,i))]
  public void addContact(String n, ContactInfo i) { ... }

  /*@ [hasContact(n) -> cset := cset->
    @    select(c: Contact | !c.getName().equals(n))] @*/
  public void removeContact(String n) { ... }

  /*@ [hasContact(n) -> result := cset->
    @     any(c: Contact | c.getName().equals(n))]  @*/
  public Contact getContact(String n) { ... }
}
```

The public specification is for clients and the private specification is for an implementor. The private specification needs to be proved to be a correct implementation or refinement of the public specification, and this is done using the abstraction function to coerce a concrete value to an abstract value.

How does the use of a model variable support a modular specification and verification of client code? If the concrete representation of a class is changed, one only needs to redefine the abstraction functions of the model variables of the class. The public specifications of the class remain the same as they are written in terms of model variables. This means that if client code is specified and verified using the public specification of the class, it is still valid and doesn't require re-specification or re-verification. Model variables also support a separation of concerns when developing a program. Once the public interface and its specification of a class are defined and formally written, the development of the class and its clients–code along with its detailed specification and verification–can be done separately and independently.

```
class GroupedAddressBook extends AddressBook {

  private Map<String,Set<Contact>> groups;

  /*@ public CJMap<String,CJSet<Contact>> cmap
    @   = CJMap.convertFrom(groups);  @*/

  /*@ [cset, cmap :=
    @   new CJSet<Contact>(), new JMap<String,CJSet<Contact>>()] @*/
  public GroupedAddressBook() { ... }

  //@ [result := cmap.containsKey(n)]
  public boolean hasGroup(String n) { ... }

  //@ [!hasGroup(n) -> cmap := cmap.put(n, new CJSet<Contact>())]
  public void createGroup(String n) { ... }

  //@ [hasGroup(n) -> result := cmap.get(n).convertToSet()]
  public Set<Contact> getGroup(String n) { ... }

  /*@ [hasContact(cn) && hasGroup(gn)->cmap := cmap.put(gn, g.add(c))
    @  where Set<Contact> g = cmap.get(gn)
    @             Contact c = getContact(cn)] @*/
  public void addToGroup(String cn, String gn) { ... }

  /*@ also
    @ [hasContact(n) -> cmap := removeContact(getContact(n))] @*/
  public void removeContact(String n) { ... }

  /*@ public CJMap<String,CJSet<Contact>> removeContact(Contact c) {
    @   CJMap<String,CJSet<Contact>> r = cmap;
    @   for (String k: r.keySet())
    @     r = r.put(k, r.get(k).remove(c));
    @   return r;
    @ }  @*/
}
```

Figure 3: Specification of the GroupedAddressBook class

3-4 Inheritance of Specifications

In CleanJava, a subclass inherits all the properties of its superclass, including annotations such as user-defined functions, model methods, and method specifications. As an example, let us introduce a new subclass of the class AddressBook, named GroupedAddressBook. The class GroupedAddressBook allows one to organize contacts into a set of named groups. A contact can now belong to several named groups. Figure 3 shows the specification of the GroupedAddressBook class. As shown, contact groups are represented as a map, named *groups*, from group names to sets of contracts belonging to the named groups. This representation is hidden, but its abstraction, a model field named *cmap*, is visible to the client and is used in specifying the behaviors of public methods. A generic class CJMap is a standard model class providing an abstraction of a map (see Section 4-3). As a model class, it is immutable. The class has a static method named *convertFrom* that coerces a java.util.Map object to a CJMap instance, and this method is used in specifying the abstraction function for the model field *cmap*.

The specification of the constructor states that initially there is no contact and no group. This is done by specifying the value of the model fields *cset* and *cmap*. Note that the model field *cset* is inherited from the superclass and is visible in the GroupedAddressBook class.

In addition to the inherited methods, the GroupedAddressBook class introduces several additional methods such as *createGroup*, *getGroup*, and *addToGroup* to manipulate contact groups. As expected, the behaviors of these methods are specified abstractly in terms of the model field *cmap*.

Perhaps, the most interesting aspect of the GroupedAddressBook class is its specification of the overriding method *removeContact*. The *removeContact* method is overridden because if a contact is removed from an address book, all its occurrences in contact groups must be removed too. Remember that the fact that a contact is removed from an address book is specified in the annotation of the overridden method in the superclass. However, this annotation is inherited to the overriding method in the subclass and thus doesn't have to be re-specified. The keyword *also* provides a visual cue that a specification is being inherited from a superclass. In short, the annotation for the overriding method *removeContact* in the subclass specifies only the fact that all occurrences of the contact are removed from contact groups, but due to specification inheritance its complete and effective specification is:

```
[hasContact(n) -> cset, cmap := cset->
   select(Contact c| !c.getName().equals(n)), removeContact(getContact(n))]
```

stating that the contact is completely removed from the address book by deleting it from both the set of known contacts, *cset*, and the contact groups, *cmap*.

4- OTHER FEATURES OF CLEANJAVA

In this section, we describe some other interesting features of CleanJava, a long with discussions of the problems and our solutions. In particular, we discuss equality of objects, composition of annotations, and built-in standard library.

4-1 Object Equality

In Java, there are two types of equality tests possible for objects, *referential semantics* and *value semantics* [9]. In referential semantics, we take a simple view of object equality and just test if two objects are the same instance—e.g., share the same address in memory—which is often referred to as an object identity. If the objects being compared are the same instance, they are considered equal. If they are not the same instance, they are considered not equal. However, many times the test for equality between two objects is not about referential semantics, but value semantics. In other words, equality may mean more about the objects having the same field values and not that they are the same instance. For this value semantics, the java.lang.Object class defines an *equals* method to be overridden by a subclass [9].

When writing intended functions, we sometimes need to make this fine distinction between referential and value semantics. The concurrent assignment introduced in previous sections uses value semantics. As an example, consider a concurrent assignment [*x* := *y*], where *x* and *y* are reference variables denoting objects. It means that the final value of *x* is equivalent to the initial value of *y* in value semantics, i.e., *x.equals*(*y*). It doesn't mean that *x* is the same instance as *y*, i.e., *x* == *y*. Thus, both *x* = *y* and *x* = *y.clone()* are correct with respect to the intended function [*x*:= *y*].

CleanJava provides a variation of the concurrent assignment for writing intended functions using referential semantics, as shown below.

[L_1, L_2, ..., L_n &= E_1, E_2, ..., E_n]

where L_i is an expression denoting a location and E_i is an expression denoting a value. For this concurrent assignment to be well-formed, (a) each L_i denotes a location, (b) the numbers of L_i's and E_i's must be the same, and (c) E_i must be assignment-compatible with L_i. One type of values is *assignment-compatible* with another type of values if a value of the first type can be assigned to a variable of the second type (refer to [9] for the rules of assignment compatibility in Java). In addition, all L_i's must be of reference types.

Now, given an intended function [x &= y], the statement x = y is correct while x = y.clone() is not, provided that the *clone* method create a new object of the same state.

4-2 Splitting and Composing Annotations

When writing an intended function, it is often convenient to split its definition by the state variables being changed. One state variable can be considered at a time, and its state change can be expressed independently from other state changes. For example, instead of writing [*x, y* := e_1, e_2], one can write [*x* := e_1, *y* := e_2]. It is particularly useful for specifying an intended function that has multiple state variables in the left hand side or of which expressions in the right hand side are long. The resulting specification is often better presented and thus is more readable. For example, the following intended function states that the new values of *r*, *c*, and *i* are, respectively, the sum of all elements of *a*, the number of positive values contained in *a*, and an arbitrary value.

```
[ r := a->iterate(int e, int m = a[i]; e > m ? e : m),
  c := a->select(int e; e > 0).size (),
  i := anything]
```

Since the above notation is a syntactic sugar, or a shorthand notation, all the expressions in the left-hand sides are evaluated concurrently in the initial state. That is, [x_1 := e_1, x_2 := e_2, ..., x_n := e_n] is equivalent to [x_1, x_2, ..., x_n := e_1, e_2, ..., e_n], and thus all e_i are evaluated at the same time in the initial state. For example, consider an intended function [x := $x + y$, y := $x - y$]. If the initial values of x and y are 10 and 20, respectively, their final values will be 30 and -10, not 30 and 10; that is, $x - y$ is also evaluated in the initial state, not in the state where x gets its new value, 30. The most general form of this new notation has the following structure.

[B_1 -> A_1, B_2 -> A_2, ..., B_n -> A_n]

where B_i is an optional boolean expression, and A_i is a concurrent assignment. As shown, each concurrent assignment can have an optional condition and multiple state variables. In syntax and structure, it is similar to the conditional concurrent assignment notation; the only difference is the use of "," or "/" as a separator. In fact, both have the same meaning if the conditions B_i's are mutually exclusive. However, if B_i's are not mutually exclusive and more than one condition hold, one is chosen non-deterministically among all the conditions that hold; remember that, for the conditional concurrent assignment, chosen deterministically is the first condition that holds. Because of this, we call it a *non-deterministic conditional concurrent assignment*. As an example, consider the following two intended functions.

f_1: [x > 0 -> z := x / y > 0 -> z := y]
f_2: [x > 0 -> z := x, y > 0 -> z := y]

The two functions denote the same function if at most one of x and y are positive; if both x and y are zero or negative, both functions are undefined. However, if both x and y are positive, f_1 maps z to x while f_2 maps z to either x or y non-deterministically; f_1 is deterministic, but f_2 is not.

We just showed how to split the definition of an intended function by specifying its mappings individually and combining them. It is also convenient to define an intended function by composing other intended functions. CleanJava provides a notation for doing this. For example, f_1; f_2 denotes a *sequential composition* of two intended functions f_1 and f_2, and its meaning is $(f_1; f_2)(x) \equiv f_2(f_1(x))$. The function f_1 is first evaluated in the initial state, and then in the resulting intermediate state the function f_2 is evaluated to produce the final state. As a shorthand notation, one can also write [f_1; f_2] instead of [f_1]; [f_2], where f_1 and f_2 are intended functions without the enclosing square bracket symbols ([]). Let's reconsider the previous examples along with a new intended function, f_3, written using the sequential composition notation.

f_1: [x > 0 -> z := x / y > 0 -> z := y]
f_2: [x > 0 -> z := x, y > 0 -> z := y]
f_3: [x > 0 -> z := x; y > 0 -> z := y]

We already know the mathematical functions that f_1 and f_2 define, but how about f_3? What function does f_3 define? Let's do a case analysis. If *x* is not positive, the function is undefined; the first component function is partial, defined only when *x* is positive. If both *x* and y are positive, it defines [*z* := *y*]; otherwise, the function is undefined because the second component function is partial defined only when *y* is positive. In short, f_3 is equivalent to [*x* > 0 && *y* > 0 -> *z* := *y*].

4-3 Standard Library

As hinted in previous sections, CleanJava provides several standard library classes for manipulating values and writing abstract specifications (see Section 3-3 for writing abstract specifications). These library classes provide abstractions of mathematical structures such as sets, bags, sequences, and mappings and define operations similar to those of Java collection classes. However, since they are intended for use in writing annotations, they are all immutable types; there is no method for changing the state of an object. For example, an *add* method creates and returns a new collection object instead of mutating the receiver object. The standard library classes are automatically imported to every CleanJava file.

The standard library classes consist of one abstract class and four concrete classes, and they are all generic classes.

- CJCollection<E>: This is an abstract superclass of other collection classes and represents a group of objects.
- CJBag<E>: This is a concrete subclass of the CJCollection class and represents an unorderd collection that may contain duplicate elements.
- CJSet<E>: This is a concrete subclass of the CJCollection class and represents an unordered collection that contains no duplicate elements.
- CJSequence<E>: This is a concrete subclass of the CJCollection class and represents an ordered collection that may contain duplicate elements. The elements may be accessed by their integer index, the position in the sequence.
- CJMap<K,V>: This class represent an object that maps keys to values. It cannot contain duplicate keys, and thus each key can map to at most one value.

In addition to the standard library classes, programmers can also define their own immutable library classes and import specifically for writing CleanJava annotations and not for use in Java code.

5- RELATED WORK

The design of CleanJava was influenced by several formal specification notations and languages. Below we summarized some of the most influencing and closely related work.

Although the foundation of our work is Cleanroom [23] [26] [27], we took many ideas from recent advances in formal specification languages such as assertions [8] [28], design-by-contract (DBC) [20] [21] [22], and behavioral interface specification languages (BISL) [16] [34]. As in Cleanroom, intended functions are written in concurrent assignment statements, however, following the idea of DBC, Java expressions are used to write intended functions. This makes it easy for Java programmers to learn and write intended functions since it minimizes the overhead of learning a separate specification notation [6].

Extensions to the Java expression syntax, such as iteration operations on arrays and collections, were inspired by the Object Constraint Language (OCL) [33]. The design of built-in mathematical toolkit including sets and sequences was based on those of Z [30], VDM-SL [13], and JML [1] [16] [17]. The syntax and semantics of user-defined mathematical functions were influenced by modern functional programming languages such as SML [24] and Haskell [12] and their integrations with object-oriented programming languages, e.g., Scala [25]. For example, if function signatures are left out, they are automatically inferred at compile time, and pattern matching can be used to split the definition of a function.

The JML language [1] [16] [17], a BISL for Java, had a great influence on the design of CleanJava. The notion of model methods and the idea of combining formal and informal texts in the specification are from JML. JML also inspired the design of CleanJava on supporting abstract and modular specifications, especially the notions of model variables [5]. Model variables and privacy of specifications support the dual uses of method specifications—verifications of both client code and the method implementation itself.

The notion of behavioral subtyping plays a key role in modular reasoning and verification of object-oriented programs [14] [15] [19], and the inheritance of specifications in CleanJava supports a behavior notation of subtyping. In CleanJava, the verification of client code in the presence of subclassing is essentially the same as in procedural programs. For each subclass, however, one has to prove that it is a behavioral subtype of its superclass by showing that each overriding method behaves like the overridden method [19]. This approach reflects the way programmers reason informally about object-oriented programs, in that it allows them to use static type information, which avoids the need to consider all possible runtime subtypes [14] [15]. In Clean-

Java, a behavioral notion of subtyping is defined in terms of intended functions for functional verification [4].

The only published work that we found on extending Cleanroom-style functional specifications for object-oriented programs is that of Ferrer [7]. Ferrer proposed to specify the behavior of a class in object-oriented programs by writing the intended functions of mutation methods in terms of the observer methods of the class. However, such a specification has an algebraic flavor and will be inherently incomplete because the intended function of the observer methods themselves can't be written. In CleanJava, a complete specification can be written by referring to and manipulating abstract values represented by model variables, and it has a flavor of model-oriented specifications.

6- CONCLUSION

We described the key features of the CleanJava language, a formal annotation language for the Java programming language, to support functional program verification. In CleanJava, annotations such as intended functions are written in the Java expression syntax extended with features from recent advances in formal specification notations and languages, such as informal descriptions, iteration operations, user-defined mathematical functions, model methods, model variables, and specification inheritance. The CleanJava language is currently being evaluated and refined through case studies, and its support tools are being developed.

ACKNOWLEDGMENT

This work was supported in part by U.S. NSF grant DUE-0837567.

REFERENCES

[1] R. J. Back and J. Wright, *Refinement Calculus: A Systematic Introduction,* Springer, 1998.

[2] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, “An overview of JML tools and applications,” *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[3] Y. Cheon, “Abstraction in assertion-based test oracles,” *The 7th International Conference on Quality Software, Portland, Oregon, USA, October 11-12, 2007*, pages 410–414. IEEE Computer Society, October 2007.

[4] Y. Cheon, *Functional Specification and Verification of Object-oriented Programs*, Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 10-23, August 2010.

[5] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, “Model variables: Cleanly supporting abstraction in design by contract,” *Software—Practice & Experience*, vol. 35, no. 6, pp. 583–599, May 2005.

[6] C. N. Dean and M. G. Hinchey, editors, *Teaching and Learning Formal Methods*. Academic Press, 1996.

[7] G. J. Ferrer, “Teaching Cleanroom software engineering with objecto-riented data abstraction,” *Journal of Computer Sciences in Colleges*, vol. 21, no. 5, pp. 155–161, 2006.

[8] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka, “Overview and evaluation of constraint validation approaches in Java,” *ICSE ’07: Proceedings of the 29th International Conference on Software Engineering*. IEEE ComputerSociety, 2007, pp. 313–322.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*. Addison-Wesley, third edition, 2005

[10] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, 12(10):576–580,583, October 1969.

[11] C.A. R. Hoare, “Proof of correctness of data representations,” *Acta Informatica*, 1(4):271–281, 1972.

[12] G. Hutton, *Programming in Haskell*, Cambridge University Press, 2007.

[13] C. B. Jones, S*ystematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[14] G. T. Leavens, “Modular specification and verification of object-oriented programs,” *IEEE Software*, 8(4):72–80, July 1991.

[15] G. T. Leavens and W. E. Weihl, “Specification and verification of object-oriented programs using supertype abstraction,” *Acta Informatica*, 32(8):705–778, November 1995.

[16] G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary design of JML: A behavioral interface specification language for Java,” *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[17] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, “How the design of JML accommodates both runtime assertion checking and formal verification,” *Science of Computer Programming*, 55(1-3):185–208, March 2005.

[18] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development,* MIT Press, Cambridge, Mass., 1986.

[19] B. Liskov and J. M. Wing, “A behavioral notion of subtyping,” *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[20] J. McKim and R. Mitchell, *Design by Contract by Example*, Adisson Wessley, 2001.

[21] B. Meyer, *Object-Oriented Software Construction,* Prentice Hall, New York, NY, second edition, 2000.

[22] B. Meyer, “Applying “design by contract”,” *Computer*, vol. 25, no. 10, pp. 40–51, October 1992.

[23] H. D. Mills, M. Dyer, and R. Linger, “Cleanroom software engineering,” *IEEE Software*, vol. 4, no. 5, pp. 19–25, September 1987.

[24] R. Milner, Tofte M, R. Harper, and D. MacQueen, *The Definition of Standard ML*, MIT Press, 1997.

[25] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima, 2008.

[26] R. Oshana, “Tailoring Cleanroom for industrial use,” *IEEE Software*, vol. 15, no. 6, pp. 46–55, November 1998.

[27] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering*, Addison Wesley, February 1999.

[28] D. S. Rosenblum, “A practical approach to programming with assertions,” *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[29] A. E. Sobel, “Emphasizing formal analysis in a software engineering curriculum,” *IEEE Transactions on Education*, 44(2), May 2001.

[30] J. M. Spivey, *Understanding Z: a Specification Language and its Formal Semantics*, Cambridge University Press, New York, NY, 1988.

[31] A. M. Stavely, *Toward Zero Defect Programming*, Addison-Wesley, 1999.

[32] A. M. Stavely, “High-quality software through semiformal specification and verification,” *CSEET ’99: Proceedings of the 12th Conference on Software Engineering Education and Training*, pages 145–155, IEEE Computer Society, 1999.

[33] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, second edition, Addison-Wesley, 2003.

[34] J. M. Wing, “Writing Larch interface language specifications,” *ACMTransactions on Programming Languages and Systems*, 9(1):1–24, January 1987.