CleanJava: A Formal Notation for Functional Program Verification

Yoonsik Cheon, Cesar Yeep and Melisa Vela

TR #10-49
November 2010; revised January 2011

Keywords: formal specification; formal verification; functional program verification; intended function; CleanJava.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Correctness proofs, formal methods; D.3.3 [*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.

To appear in the *8th International Conference on Information Technology (ITNG 2011), April April 11-13, 2011, Las Vegas, Nevada.*

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

CleanJava: A Formal Notation for Functional Program Verification

Yoonsik Cheon, Cesar Yeep, and Melisa Vela
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
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 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 paper, we propose 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—formal specification; formal verification; functional program verification; intended function; CleanJava

I. Introduction

In the late 70s, Harlan Mills and his colleagues at IBM developed an approach to software development called *Cleanroom Software Engineering* [1] [2]. 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* [3] [4]. 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* [4]. 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 programmers reason about the correctness of a program informally by supporting forward reasoning. 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 such as Hoare logic 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.

In this paper we propose 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) extensible vocabularies through user-defined functions and specification-only methods, (b) a wide-spectrum of formality that can be tuned, (c) support for abstraction and modularity, and (d) support for object-oriented concepts such as specification inheritance. We believe that CleanJava is a good notation for writing intended functions and facilitates formal correctness verification and reasoning of Java programs.

The rest of this paper is organized as follows. Section II provides an overview of the functional program verification. Section III describes the core part of the CleanJava language, including its syntax for writing intended functions. Section IV explains mechanisms for introducing user-defined vocabularies for writing intended functions. Section V explains an approach for writing abstract specifications to support modular specification and verification. Section-VI describes the inheritance of specifications. Section VII discusses related work, and Section VIII concludes this paper.

II. Functional Program Verification

A. 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 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; 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. 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 doesn't 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*.

B. 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 such that:

- The domain of p is a superset of the domain of f, i.e., $\text{dom}(p) \supseteq \text{dom}(f)$.
- 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 $x \in \text{dom}(f)$.

We also say that p is a *refinement* of f, denoted by $p \sqsubseteq f$.

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

```
//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 largest value 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. In function f_2, a question mark symbol (?) 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 to discharge 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.
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

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

$$
\begin{array}{lcl}
f_1; f_2 & \equiv & [r, i := a[0], 1]; \\
 & & [r, i := \text{max of r and largest value in a[i..]}, ?] \\
 & \equiv & [r, i := \text{max of a[0] and largest value in a[1..]}, ?] \\
 & \equiv & [r, i := \text{largest value in a}, ?] \\
 & \sqsubseteq & [r := \text{largest value in a}] \\
 & \equiv & f_0
\end{array}
$$

In functional verification, the proof is sometimes straightforward because we can calculate functions and compare them. However, we 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.

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

III. The Core Language of CleanJava

CleanJava is a formal notation for annotating Java code with intended functions. It supports rigorous or formal verification of Java code. In this section, we describe the core part of the CleanJava language.

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 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, (* largest 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 enclosed in `//@` or `/*@ ... @*/`. 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 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 several CleanJava-specific operations defined on arrays and collections. It has a general form of `iterate`(T_1 x, T_2 $y \mid E(x)$), 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. Other iteration operations defined on arrays and collections include `select`, `reject`, `collect`, `forAll`, and `exists`.

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.

IV. Extension Mechanisms

One feature of CleanJava is that its vocabulary is 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.

A. User-defined Functions

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 and the informal description removed.

```
//@ 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 `sum` 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 functional languages such as SML and Haskell, they are inferred statically at compile time. A CleanJava function follows the Java scoping rules. 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 in Section IV-B).

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.

B. Model 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*. 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 to manipulate 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

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

Fig. 1. An example annotation written using a model method

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. 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 `addContact` method is partial in that its behavior is defined only when there exists no contact in the address book with the given name (`n`). The optional condition preceding the arrow symbol (→) specifies the domain of the intended function; if omitted, the intended function is a total function.

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; 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.

V. Support for Abstraction

CleanJava provides several features to support modular specification and verification. Verification of client code of a class is said to be *modular* if a change on the the hidden implementation details of the class such as data structures and algorithms doesn't require a re-verification of the client code. For 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 can be tightly coupled to the class by exploiting an exposed implementation detail or decision of the class. Its verification or reasoning will not be modular either 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. 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 can 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*, 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` of type `CJSet<Contact>`. 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`. However, one key difference from the `Set` interface is that it is an immutable type because it is supposed to be used in CleanJava annotations. 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* and is specified in an optional initializer of a model variable declaration. For example, the value of a model 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 func-

```
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) { ... }
}
```

Fig. 2. Revised specification of the AddressBook class

tions 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;
  }
```

The public specification is for clients and the private specification 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 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 CJMap<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;
    @*/
}
```

Fig. 3. Specification of the GroupedAddressBook class

VI. 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 interface 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. As a model class, it is an immutable type. The class has a static method named `convertFrom` that coerces a `java.util.Map` object to an `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 part 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 occurrence from contact groups also have to be removed. 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 is inherited to the overriding method in the subclass and thus doesn't have to be re-specified in the subclass. 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 only specifies the fact that all occurrences of the contact removed from the address book should also be 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))]
```

VII. Related Work

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

Although the foundation of our work is Cleanroom [1] [2] [5], we also took ideas from recent advances in formal specification languages such as design-by-contract (DBC) notations [6] and behavioral interface specification languages (BISL) [7]. As in Cleanroom, intended functions are written in concurrent assignment statements, however, following the idea of DBC, Java's expression syntax is 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. Extensions to the Java expression syntax, such as iteration operations on arrays and collections, were inspired by the Object Constraint Language (OCL) [8]. The design of built-in mathematical toolkit including sets and sequences was based on that of Z, VDM-SL, and JML [7]. The syntax and semantics of user-defined mathematical functions were influenced by modern functional programming languages such as SML and Haskell and their integrations with object-oriented programming languages, e.g., Scala [9]. The JML language, a BISL for Java, had a great influence on the design of CleanJava [7]. The notion of a model method and the idea of combining formal and informal texts in the specification of an intended function are from JML. JML also inspired the design of CleanJava on supporting abstract and modular specifications, especially the notions of model variables [10]. Model variables and privacy of specifications support the dual uses of a method specification—verifications of both client code and the method implementation itself.

The only published work that we found on extending Cleanroom-style functional specifications for object-oriented programs is that of Ferrer [11]. 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.

VIII. Conclusion

We described the key features of CleanJava, 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 of formal specification 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 by NSF grant DUE-0837567.

References

[1] H. D. Mills, M. Dyer, and R. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19–25, Sep. 1987.

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

[3] 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, Aug. 2010, to apper in the *Annual International Conference on Software Engineering (SE 2010)*, December 6-7, 2010, Phuket, Thailand.

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

[5] R. Oshana, "Tailoring Cleanroom for industrial use," *IEEE Software*, vol. 15, no. 6, pp. 46–55, Nov. 1998.

[6] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[7] 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.

[8] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison-Wesley, 2003.

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

[10] 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.

[11] G. J. Ferrer, "Teaching Cleanroom software engineering with object-oriented data abstraction," *Journal of Computer Sciences in Colleges*, vol. 21, no. 5, pp. 155–161, 2006.