5-2011

Functional Verification of Class Invariants in CleanJava

Carmen Avila
The University of Texas at El Paso, ceavila3@miners.utep.edu

Yoonsik Cheon
The University of Texas at El Paso, ycheon@utep.edu

Comments:

Technical Report: UTEP-CS-11-26

Functional Verification of Class Invariants in CleanJava

Carmen Avila and Yoonsik Cheon Cheon

TR #11-26
May 2011; revised November 2011

Keywords: class invariant, functional program verification, intended function, proof logic, CleanJava.

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods; D.3.3 [*Programming Languages*] Language Constructs and Features — Classes and objects; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

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

Functional Verification of Class Invariants in CleanJava

Carmen Avila and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
ceavila3@miners.utep.edu; ycheon@utep.edu

***Abstract*—In Cleanroom-style functional program verification, a program is viewed as a mathematical function from one program state to another, and the program is verified by comparing two functions, the implemented and the expected behaviors of a program. The technique requires a minimal mathematical background and supports forward reasoning, but it doesn't support assertions such as class invariants. However, class invariants are not only a practical programming tool but also play a key role in the correctness proof of a program by specifying conditions and constraints that an object has to satisfy and thus defining valid states of the object. We suggest a way to integrate the notion of class invariants in functional program verification by using CleanJava as a specification notation and a verification framework as well; CleanJava is a formal annotation language for Java to support Cleanroom-style functional program verification. We propose a small extension to CleanJava to specify class invariants and to its proof logic to verify the class invariants. Our extension closely reflects the way programmers specify and reason about the correctness of a program informally. It allows one to use class invariants in the framework of Cleanroom-style functional specification and verification.**

Keywords: class invariant, functional program verification, intended function, proof logic, CleanJava.

I. Introduction

An assertion is a predicate or boolean expression, placed in a program, that should be always true at that place [1]. Assertions such as class invariants and operation pre- and post-conditions became popular as a practical programming tool for verifying, testing and debugging programs [2]. If an assertion evaluates to false at runtime, it indicates that there is an error in the code for that particular execution, thus an assertion can be used for runtime verification of code and for narrowing down a problematic part of the code. Assertions also play a key role in verifying statically the correctness of a program [3]. A class invariant, for example, specifies a condition that all objects of a class must satisfy while they can be observable by clients. It defines valid states of an object and ensures that an object remains in a consistent state. It must be proved that all methods of the class preserve the class invariant.

A functional program verification technique such as Cleanroom [4] views a program as a mathematical function from one program state to another and proves its correctness by essentially comparing two functions, the function computed by the program and its specification [5]. Since the technique uses equasional reasoning based on sets and functions, it requires a minimal mathematical background. Unlike Hoare logic [1], it also supports forward reasoning and thus reflects the way programmers reason about the correctness of a program informally. There is a formal notation to support Cleanroom-style functional program verification. CleanJava is such a formal annotation language for the Java programming language [6]. In CleanJava, a specification function is written using a subset of Java expressions enriched with CleanJava-specific extensions, and every section of Java code is annotated with its expected behavior for formal verification of the correctness of the code (see Section II).

One problem of a functional program verification technique, however, is that it doesn't work well with assertions, especially with class invariants. In fact, CleanJava doesn't provide any built-in language construct to express class invariants. This poses a serious problem both in writing a specification and using it for a correctness proof. In CleanJava, for example, the behavior of a method is specified as a mathematical function, and thus a class invariant must be expressed in a functional form and merged to the specification of each method of the class. The resulting specifications become less readable, reusable, and maintainable, and the correctness verification is not modular in that it can't be decomposed into those of an invariant property and a method-specific property.

In this paper we propose a way to integrate the notion of class invariants in the functional program verification by using CleanJava as a platform for our study. We suggest two approaches: an *invariant function* and an *invariant clause.* In the first approach, a user-defined function is introduced to test a class invariant. This invariant function is referred to in the specification of each method of a class. The second approach supports an invariant as a built-in language feature by extending CleanJava and its proof logic. It adds a special clause to express an invariant of a class and extends the proof logic to ensure that the specified invariant be established by constructors and preserved by all methods of the class. Although this approach requires a language extension, it provides a better solution by cleanly separating the specification and verification of an invariant from those of methods.

Since invariants are a well-known concept, it isn't surprising to find existing work on using invariants in a Cleanroom-style verification [5]. However, the topic's treatment is shallow in an informal setting without giving a systematic way of translating an invariant or a formal treatment of its proof rules.

The main contribution of our work is that it enables one to use class invariants in the framework of a Cleanroom-style functional specification and verification technique and thus makes the technique more closely resemble the way

```
class AddressBook {
  private List<Contact> db;

  //@ [db := new ArrayList<Contact>()]
  public AddressBook() {
    db = new ArrayList<Contact>();
  }

  /*@ f0:[n != null ->
          result := db->exists(getName().equals(n))] @*/
  public boolean hasContact(String n) {
    //@ f1: [r, i := false, 0]
    boolean r = false;
    int i = 0;

    /*@ f2:[r, i := r || b, anything] where
             boolean b = db.subList(i, db.size())
                          ->exists(getName().equals(n)) @*/
    while (i < db.size()) {
     //@ [r, i := r || db.get(i).getName().equals(n), i++]
       if (contacts.get(i).getName().equals(n))
          r = true;
       i++;
    }

    //@ f3:[result := r]
    return r;
  }
}
```

Fig. 1. Sample CleanJava code

programmers specify and reason about the correctness of a program informally. We expect this to have a positive effect on teaching and practicing the functional program verification.

The rest of this paper is structured as follows. In Section II below we give a quick overview of CleanJava and functional program verification. In Section III we illustrate the problem of the functional verification not supporting class invariants. In Section IV we describe our two approaches for integrating the notion of invariants in a functional verification technique, followed by a comparison of these approaches. In Section V we provide a concluding remark along with future work.

II. Background—CleanJava

CleanJava is a formal annotation language for the Java programming language to support a Cleanroom-style functional program verification [6]. In the functional program verification, a program is viewed as a mathematical function from one program state to another. In essence, functional verification involves calculating the function computed by code, called a *code function*, and comparing it with the intention of the code written as a function, called an *intended function* [5]. CleanJava provides a notation for writing intended functions. A *concurrent assignment* notation, $[x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$, is used to express these functions by only stating changes that happen. It states that x_i's new value is e_i, evaluated concurrently in the initial state—the state just before executing the code; the value of a state variable that doesn't appear in the left-hand side remains the same. For example, $[x, y := y, x]$ is a function that swaps two variables x and y.

Figure 1 shows sample Java code annotated with intended functions written in CleanJava. It describes an AddressBook class containing a collection of contacts. A CleanJava annotation is written in a special kind of comments either preceded by `//@` or enclosed in `/*@ ... @*/`, and an intended function is written in the Java expression syntax with a few CleanJava-specific extensions. The first annotation states that the constructor initializes the `db` field to a new empty list. The intended function of the `hasContact` method is interesting. It specifies a partial function defined only when the argument (`n`) is not null; as shown, a concurrent assignment may have an optional condition or guard followed by an arrow (`->`) symbol. The function states that, given a non-null name (`n`), the method tests if there is a contact with the given name in `db`. The pseudo variable `result` denotes the return value of a method, and `exists` is a CleanJava iteration operator that tests if a collection contains at least one element that satisfies a given condition. The body of the method is also interesting. Each section of code is documented with its intended function. In the function f_2, the keyword **anything** indicates that we don't care about the final value of the loop variable `i`, and a **where** clause introduces local definitions such as that of `b`.

It would be instructive to sketch a correctness proof of the `hasContact` method, which involves the following.

- Proof that the composition of functions f_1, f_2, and f_3 is correct with respect to ($\sqsubseteq$), or a refinement of, f_0, i.e., $f_1; f_2; f_3 \sqsubseteq f_0$, where ; denotes a functional composition.
- Proof that f_1, f_2, and f_3 are correctly refined

In the functional verification, a proof is often trivial or straightforward because a code function can be easily calculated and directly compared with an intended function; e.g., f_1 and f_3 are both code and intended functions. However, one also 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 [6]. Below we discharge the first proof obligation, where b_i is $db.subList(i, db.size()) \rightarrow exists(getName().equals(n))$ and ? is short for **anything**.

$$
\begin{aligned}
f_1; f_2; f_3 &\equiv [r, i := false, 0]; [r, i := r || b_i, ?]; [result := r]; \\
&\equiv [r, i := b_0, ?]; [result := r]; \\
&\equiv [r, i, result := b_0, ?, b_0] \\
&\sqsubseteq [result := b_0] \\
&\equiv f_0
\end{aligned}
$$

III. The Problem

A functional program verification technique is fundamentally different from an assertion-based technique such as Hoare logic [1]. It is direct and constructive in that for each state variable such as a program variable one must state its final value explicitly. On the other hand, an assertion-based technique is indirect and constraint-based in that one specifies the condition that the final state has to satisfy by stating a relationship among state variables. The final value of a state variable isn't defined directly but instead is constrained and given indirectly by the specified condition.

Because of this fundamental difference, a functional verification technique doesn't work very well with assertions such

as class invariants. In fact, CleanJava doesn't provide a built-in language construct for specifying class invariants. This is a serious concern in practice because class invariants are a popular programming idiom and can't be directly expressed in CleanJava. To illustrate this problem, let's consider the AddressBook class from the previous section. One possible class invariant for this class would be *db != null && db→isUnique(getName())*, stating the non-nullness of the db field and the uniqueness of contact names; the isUnique operator is a CleanJava iterator asserting the uniqueness of given values. How to express this invariant in CleanJava? An invariant must be merged to, and expressed in, the intended function of each operation of the class to ensure its establishment by a constructor and its preservation by each method, as shown below.

```
class AddressBook {
  private List<Contact> db;

  /*@ [db := new ArrayList<Contact>] @*/
  public AddressBook() { ... }

  /*@ [n != null && db != null && db->isUnique(getName()) ->
       result := db->exists(getName().equals(n))] @*/
  public boolean hasContact(String n) { ... }
}
```

Note that the constructor's intended function remains the same. This is because the new value for db, an empty list, obviously implies the invariant. For the hasContact method, the invariant becomes the optional condition part of the concurrent assignment and the rest are unchanged. This is because a method assumes an invariant, and this particular method doesn't change any state variable, meaning that the invariant is trivially preserved. For a mutation method, say addContact, the invariant must become the condition of its intended function and be implied by new values of state variables.

There are several shortcomings in the above approach of not explicitly stating a class invariant and scattering it all over method specifications. There are problems of specification readability, reusability, and maintainability. The specifications of an invariant property and the behavior of a method are tangled, and an invariant specification is duplicated in almost every method specification. The approach also makes a correctness verification hard and non-modular in that the verification of an invariant property and that of a method-specific property can't be performed separately, as the specifications of these two properties are tangled and are not distinguished.

IV. Our Approach

In this section we describe our approaches for supporting invariants. We propose two approaches: an invariant function and an invariant clause. The first approach allows one to systematically translate an invariant to CleanJava annotations without requiring an extension to CleanJava or its proof logic. On the other hand, the second approach does require an extension to both the notation and the proof logic of CleanJava, but it cleanly separate the specification and verification of class invariants from those of methods.

A. An Invariant Function

This approach is to express an invariant in the intended function of each method. An invariant becomes part of the intended function of a method and is verified along with the intended function. This approach is similar to the view that an invariant is conjoined to pre- and post-conditions of an operation. To eliminate a duplication of an invariant expression in multiple intended functions, we introduce a user-defined function that tests an invariant. This function is called an *invariant function* and is responsible for testing all the invariants of a class.

Suppose we have an invariant I written in terms of a state variable, say x, and a method with an intended function $[P \to y := E]$, where the type of y is T. Then, our approach produces the following user-defined function and intended functions.

```
fun inv(x) = I
f1:[inv(x) && P -> y := E]
f2:[inv(x) && P ->
    y := findAny(T z|z == E && inv(z))]
```

The first annotation introduces a user-defined function named inv that tests the invariant of a class. The state variables appearing in the invariant become the arguments of the invariant function so that the invariant can be tested in both the initial and the final states. The next two annotations show translated intended functions. Depending on whether a state variable appearing in the invariant is changed or not, either intended functions f_1 or f_2 is used. If x and y are different state variables—i.e., the state variable appearing in the invariant is not changed, the first one (f_1) is used; otherwise, the second one (f_2) is used. As expected, the invariant constrains the condition (P) and the final values of state variables (E). The CleanJava operator findAny denotes an arbitrary value that satisfies a given condition. In f_2, the argument to the second inv call is z—the final value of y—because the expressions in concurrent assignments are evaluated in the initial state and the inv call is to check the invariant in the final state.

Let's apply this approach to our AddressBook class. The revised intended functions are shown below.

```
class AddressBook {
  private List<Contact> db;
  //@ fun inv(db) = db != null && db->isUnique(getName())

  /*@ [db := findAny(List<Contact> l | inv(l) &&
       l.equals(new ArrayList<Contact>()))] @*/
  public AddressBook() { ... }

  /*@ [n != null && inv(db) ->
       result := db->exists(getName().equals(n)))] @*/
  public boolean hasContact(String n) { ... }
}
```

An invariant function is defined in the first annotation. In CleanJava, one doesn't have to declare the signature of a user defined function; it is inferred [6]. The constructor's intended function was translated using the f_2 pattern. However, since a constructor doesn't assume an invariant in the initial state, the invariant function doesn't appear in the optional condition

part of the concurrent assignment. The intended function of the `hasContact` method is translated using the f_1 pattern.

B. An Invariant Clause

This approach is to support the notion of invariants as a built-in language feature of CleanJava. For this, we propose to introduce a new CleanJava language construct called an *invariant clause*. An invariant clause can appear only in the member declaration level and specifies the invariant of a class. It must be established by all constructors and preserved by all methods of the class. For example, shown below is the AddressBook class annotated using an invariant clause. Note that the intended functions of the constructor and the method are unchanged.

```
class AddressBook {
  private List<Contact> db;
  //@ inv: [db != null && db->isUnique(getName()]

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

  /*@ [n != null ->
       result := db->exists(getName().equals(n))] @*/
  public boolean hasContact(String n) { ... }
}
```

A natural next question is how to verify a class invariant specified using an invariant clause. We extend the proof rules of CleanJava to support the invariant clause. Consider a class with an invariant I specified using an invariant clause. For a constructor C with an intended function f in the form of $[P \rightarrow x := E]$, we have the following extended proof obligations.

1) C is correct with respect to f, i.e., $C \sqsubseteq f$.
2) C establishes I. For this, one needs to prove:
   a) $P \Rightarrow I$ if I is not written in terms of x, or
   b) $P \Rightarrow I[E/x]$ otherwise, where $I[E/x]$ means I with every free occurrence of x replaced with E.

For a method M with an intended function f in the form of $[P \rightarrow x := E]$, we have the following extended proof obligations.

1) M is correct with respect to f provided that I holds in the initial state, i.e., $M \sqsubseteq [P \;\&\&\; I \rightarrow x := E]$.
2) M preserves I. For this, one needs to prove:
   a) $I \wedge P \Rightarrow I$ if I is not written in terms of x, or
   b) $I \wedge P \Rightarrow I[E/x]$ otherwise, where $I[E/x]$ means I with every free occurrence of x replaced with E.

As an example, let's prove the invariant of the AddressBook class. For the constructor, we need to discharge the proof obligation 2.b: $P \Rightarrow I[E/x]$ because the constructor changes the state variable db appearing in the invariant. Note that the constructor doesn't have an optional condition (P), leaving as a proof obligation $I[E/x]$, db != $null$ && $db{\rightarrow}isUnique(getName())$ where db is **new**$ArrayList{<}Contact{>}$. The proof is straightforward because a new empty list is not null and contains no contact. For the `hasContact` method, we have to discharge the proof obligation 2.a: $I \wedge P \Rightarrow I$, as it doesn't change any state variable. However, there is nothing to prove; it's a tautology.

C. Comparison

The invariant function approach allows one to systematically translate class invariants to intended functions. Since invariants are factored out to user-defined functions, they are not duplicated in intended functions. The strength of this approach is that it doesn't require a language extension or the proof rules. However, it doesn't completely address the original problems of readability, reusability, maintainability, and verifiability. For example, specifications are still tangled and scattered, and the use of `findAny` operator in an intended function makes a specification complicate and hard to read and understand.

An invariant clause addresses all the aforementioned problems by cleanly separating an invariant specification from method specifications. It supports a separation of concerns in a verification; an invariant verification and a method verification can now be performed separately and in a modular way. Another strength of this approach is that it can also support the inheritance of an invariant by making a subclass to inherit the invariants of its superclasses. However, the approach requires an extension to both the language and its proof rules.

V. Conclusion

We suggested two approaches for supporting class invariants in Cleanroom-style functional program verification. The first approach systematically translates class invariants to intended functions by factoring them out. It doesn't require a notational or proof logic extension but is subject to the problems of readability, reusability, maintainability, and verifiability. The second approach supports class invariants as a built-in language concept. For this, we introduced a new language construct, called an *invariant clause*, and defined its meanings in terms of proof rules. This approach addresses all the aforementioned problems associated with the first approach and closely reflects the way programmers specify and reason about the correctness of a program informally.

In our study, we assumed that state variables are independent without aliasing and one state variable is not contained or owned by another. A related question is the granularity of frame axioms that assert which objects—the whole or part?—are allowed to be changed. These are future research problems.

Acknowledgment

This work was supported in part by NSF grants CNS-0707874 and DUE-0837567.

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Com. of ACM*, vol. 12, no. 10, pp. 576–580,583, Oct. 1969.

[2] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Trans. on Soft. Eng.*, vol. 21, no. 1, pp. 19–31, Jan. 1995.

[3] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," in *ECOOP 2002, Máalaga, Spain*, ser. LNCS, vol. 2374. Springer-Verlag, Jun. 2002, pp. 231–255.

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

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

[6] Y. Cheon, C. Yeep, and M. Vela, "Cleanjava: A formal notation for functional program verification," in *ITNG 2011: 8th International Conference on Information Technology: New Generations, April 11-13, 2011, Las Vegas, NV*. IEEE Computer Society, 2011, pp. 221–226.