Specification Idioms from Industrial Experience

David R. Cok
CEA, LIST, Software Safety and Security Laboratory
Gif-sur-Yvette, France
david.r.cok@gmail.com

Abstract

The author has performed various formal verification projects on industrial software of particular interest to the sponsoring company or its customers, using the Java Modeling Language and the OpenJML tool. Such projects provide particular insight into both usability and expressiveness of formal verification languages and tools. This paper describes several specification idioms that fill particular specification needs encountered in the verification projects. Consistent style and use of idioms helps readers quickly understand the content of specifications. Useful idioms also point to opportunities for specification inference and syntactic sugar.

Keywords JML, OpenJML, formal verification, specification

ACM Reference Format:
David R. Cok. 2018. Specification Idioms from Industrial Experience. In *ACM Digital Library*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.475/123_4

1 Introduction

Verification projects on real-world code, both legacy code and software under development, are essential to assessing the expressiveness and usefulness of a specification language and tools. The author is conducting such projects, not as exercises, but to improve the confidence that target code is correct, where the code under study is of particular importance to the authors or clients of the software. In these projects the code is written in Java (Java 8); the specifications use the Java Modeling Language (JML) [1, 5] and the OpenJML verification tool [2–4] to specify and verify the target software.

In the process of multiple projects, we have found several specification idioms that are commonly used. The subsections that follow illustrate some of these. Knowing and then using such idioms makes it easier to correctly and efficiently write specifications for common situations. However, they are also useful for a more important reason: understandability. A goal of some of the verification projects is to produce and maintain specifications with the original source code (and not just as a side project, to be eventually neglected); the replaying of proofs of specifications will become part of the continuous integration and testing cycle for the software. As such the specifications will be read by developers who are not the original authors of the specifications and who may not be familiar with specifications. In our projects, the developers are also quite concerned about consistency of style and readability of the source code. That same concern translates to specifications: specifications should have a consistent textual and semantic style to aid readability and quick comprehension.

A last reason for identifying common specification idioms is as an aid in evaluating the specification language. Commonly repeated idioms are opportunities for syntactic sugar, defaults, and specification inference that can reduce the amount of specification text and improve understanding.

In line with the projects we have completed and are executing, the examples are given here in Java and JML. Some idioms may be specific to a particular programming language, but many idioms will be applicable across different programming languages. Hence, we believe this compilation is useful for specification languages in general and should serve to prompt discussion and research in the evolution of specification languages.

```
1  /*@ spec_public */ private int value;
2
3  //@ public normal_behavior
4  //@   ensures \result == value;
5  //@ pure
6  public int value( ) { return value; }
7
8  //@ public normal_behavior
9  //@   assignable this.value;
10 //@   ensures this.value == value;
11 public void set(int value) { this.value = value; }
```

Figure 1. Getter and setter functions

2 Getter and setter functions

As a simple initial example, consider the standard specifications of getter and setter functions as shown in Fig. 1. Getter and setter functions are so common that we adopt precisely the same style in each case. Doing so allows quick writing and quick recognition without even having to read in detail.

FTfJP 2018, July 2018, Amsterdam NL

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

```
1 //@ public normal_behavior
2 //@   requires i != null && i != Integer.MAX_VALUE;
3 //@   ensures \result == i + 1;
4 //@ also public exceptional_behavior
5           // Solution: use private visibility here
6 //@   requires i == null || i == Integer.MAX_VALUE;
7 //@   signals_only IllegalArgumentException;
8 //@ pure
9 public Integer increment(Integer i) {
10    if (i == null || i == Integer.MAX_VALUE)
11        throw new IllegalArgumentException(i.toString());
12    return i+1;
13 }
```

Figure 2. Checking error handling paths (program has a bug)

The getter function is pure (no side-effects) and has a result value equal to the field being gotten. The field being read must be declared `spec_public` so that it has public visibility in the specification. A setter function has just the field being set mentioned in the assignable clause and a postcondition stating the new value of the field.

This style is very standard and is verbose compared to the code being specified. Thus it is a good opportunity for a default specification produced by a specification inference tool. It would take little more than pattern matching to determine instances of getter and setter functions and provide them with a template specification.

3 Specifying error paths

Error checking code can be prone to software errors because it is less tested than the non-error aspects of an application. Static checking can be a good way to check that error recovery code does not cause further errors or violate some safety or security protocols. However, error-handling code poses a dilemma. On the one hand, a defensive programming style advocates that appropriate checks for error conditions be included in the code, along with a suitable recovery or controlled exit. On the other hand, in a correct program, such conditions should never occur and the defensive checks would be dead code. (Note that we are considering here internal defensive code that protects against incorrect use of a method, not checks that catch incorrect user input or unexpected environmental conditions.)

Consider the example in Fig. 2. Here we want to check that if the input argument has an illegal value, the given exception is thrown. In fact, this code has an error in the error handling code (if the JML default is not non_null_by_default). But the specification above allows a calling method to call `m` with either legal or illegal arguments: the effective precondition is the disjunction of the two requires clauses, which simplifies to `true`. At least in most use cases, we want the static checker to issue a warning if the method `m` is being called with an illegal argument.

We can achieve that by omitting the `exceptional_behavior` part of the specification. Then there is just one precondition; a calling method must satisfy that precondition; a warning will be issued by a static checking tool if the precondition is not satisfied, that is, if the argument has an illegal value. But with the second behavior omitted, the verification of the method `m` will not check the error path, since it presumes that the precondition is always satisfied.

So, we want the caller to see only the first behavior, but the implementation to see both behaviors. JML can accomplish this within current JML by declaring the second behavior as `private exceptional_behavior`.[1]

This is not quite a complete solution, since other methods within the same class as `m` still see the `exceptional_behavior` set of clauses. We could restrict the visibility of the `exceptional_behavior` to only its owning method if we introduced a new keyword that indicated *extra-private*, namely visibility restricted to just the owning method, and not to other methods in the same class.

To date, such an extension has not been needed for the following reason. If a calling method indeed called `m` with a possibly-illegal argument (that is, not provably legal), then there would be a possible control path that threw an exception. Then the calling method would need to either handle such an Exception explicitly or account for such a behavior in its own specification.

4 Avoiding bit-vector operations

Java, like C, allows bit-vector operations on integers along with conventional arithmetic operations. This is convenient when the integer is being used as a vector of boolean flags, but it can also be an efficient way to do some numerical calculations. For example, the Euclidean modulo operation for n a power of two is given by $a \bmod n = (a\&(n-1))$.

SMT solvers implement both bit-vector and integer operations. However, no standard SMT-LIB language currently includes both bit-vectors and integers or allows converting between them.[2] Now arithmetic operations are defined for bit-vectors, so a method that contains a mixture of bit-vector and integer operations can be translated entirely into a bit-vector logic; on the other hand such a mixture cannot be translated entirely into an integer logic. The problem is that typically (at least with current technology) proofs in bit-vector logic take very much longer, even orders of magnitude longer, than in integer logics. For 64-bit integers, the time can be hours, if successful at all. Thus it is desirable to use integer arithmetic whenever possible.

Now consider a method `mcaller`, whose implementation does not use bit-vector operations, that calls a method `mcallee` that does. The proof tool will need to use bit-vector

[1] This solution was suggested in private communication by Gary Leavens.
[2] Some individual solvers may provide this functionality, but it is not yet standard SMT-LIB.

```
1  class X {
2    /*@ spec_public */ private int N; // positive power of 2
3    /*@ spec_public*/ private int p;
4    /*@ spec_public */ private byte[] buf; // size N
5
6   //@ ...
7   //@ assignable p, buf[ p>=0 ? p%N : p%N==0 ? 0 : p%N+N];
8   void m(int k) {
9       buf[ p & (N-1) ] = ...
10      p += k;
11   }
12 }
```

Figure 3. Avoiding bit-vector operations

logic for `mcallee`, but must it do so for `mcaller` as well? Recall that the proof of `mcaller` uses only the specification of methods it calls. Thus if we write the specification of `mcallee` in integer arithmetic even though the implementation contains bit-vector operations, then `mcaller` can be proved using an integer logic. In practice, in examples we have experienced, adopting this idiom — writing specifications without bit-vector operations — makes a considerable difference in performance, at the cost of some degree of understandability.

For example, consider the (simplified) code shown in Fig. 3. Here an index p is incremented by k, but is used modulo N to write into a buffer (of size N). That modulo operation is easily computed as $p\&(N-1)$. In non-bit-vector arithmetic, the expression is equivalent to the more complex ($p >= 0?p\%N : p\%N == 0?0 : p\%N + N$). Despite this additional complexity, a specification that uses the integer logic can be more efficiently used by the SMT solver.

5 Lemmas and use directives

The idiom of the last section can be taken a step further. Sometimes a large method can have just one bit-vector operation within its body. The one bit-vector operation requires the whole body to be processed in a bit-vector logic and can result in the solver taking excessive time or not completing. However, if that one operation is replaced by a integer-logic equivalent, the method verifies easily. Changing the code for that one operation would mean changing the implementation in an unchecked, unprincipled way.

As an example, Fig. 4 demonstrates two idioms. First, the code defines a pure model method named `lemma1`. This method has an empty body. Because it has a body, a proof tool will attempt to prove it valid. That proof will simply consist of assuming the precondition and attempting to prove the postcondition. In this case, that will prove a straightforward mathematical result; this lemma, although it contains a mixture of integer and bit-vector operations, is small enough to be proved.

Then, secondly, this lemma is used in the body of `m`. Here if a bit-vector operation is used within the larger method, then the method can not be proved. Now that we have proved

```
1  class X {
2    /*@ spec_public */ private int N; // positive power of 2
3    /*@ spec_public*/ private int p;
4    /*@ spec_public */ private byte[] buf; // size N
5
6   //@ ...
7   //@ assignable p, buf[ p >= 0 ? p%N : p%N == 0 ? 0 :
    p%N + N ];
8   void m(int k) {
9       ...
10      //@ use lemma1(p, N);
11      buf[ p & (N-1) ] = ...
12      p += k;
13      ...
14   }
15
16   //@ public normal_behavior
17   //@  requires k > 0 && Integer.bitCount(k) == 1;
18   //@  ensures ( i & (k-1) ) == ( i >= 0 ? i%k : i%k == 0 ? 0 :
    i%k + k );
19   //@ model public static pure void lemma1(int i, int k) {}
20 }
```

Figure 4. Using lemmas

`lemma1`, we are justified in applying `lemma1` to replace (automatically) $p\&(N-1)$ by $p >= 0?p\%N : p\%N == 0?0 : p\%N + N$, so long as we can first prove the precondition, that $N > 0$ && $Integer.bitCount(N) == 1$. The **use** directive is an enhancement to JML added in OpenJML for this purpose. It can be used in situations other than bit-vector operations — any time we would like to do a substitution within the logic representation of the code using a result, the lemma, that can be proved in isolation. Lemmas are commonly used in automated proof (e.g., Dafny [6] has lemmas as part of the programming language), but here the application is improved performance.

6 Lemmas and libraries

Using a model method in Fig. 4 to force a proof is also useful in verifying libraries. Verifying a library has challenges different than those of verifying an application. A typical software library has a large API of methods that are expected to work together in a coherent way. For example, in an implementation of a `Stack` class, a `pop` operation is expected to undo the effect of a previous `push` operation. Now a careful development team will write specifications for each method and verify them using an automated tool, to check that the implementation of each method conforms to its specification. However, it is only by inspection that one can determine that the collection of methods forms a coherent API. Lemmas can help prove that at least some use cases of the API will function correctly.

Fig. 5 shows some examples. The first example checks that the `top` of a stack returns the value expected after a `push`. The second example shows a different style to accomplish a different test: a combination of program statements and

```
class Tests {
  //@ public normal_behavior
  //@   requires s != null;
  //@   ensures \result == i;
  /*@ model public static pure Integer
      testPushTop(Stack<Integer> s, Integer i)
           { s.push(i); return s.top(); }
   @*/

  //@ public normal_behavior
  //@   requires s != null && !s.isEmpty();
  //@   old Integer initialTop = s.top();
  //@   ensures \result == initialTop;
  /*@ model public static pure Integer
      testPushPop(Stack<Integer> s, Integer i) {
       Integer k = s.top();
       s.push(i);
       s.pop();
       return s.top();
     }
   @*/

   //@ public normal_behavior
   //@   ensures i == F.finverse( F.f( i ) );
   //@ model public static pure void testInverse(int i) { }
}
```

Figure 5. Static tests of an API

specifications. The third example is again an example of a postcondition used to establish a general property of an API.

In a sense, tests such as these are similar to dynamic, runtime tests. They check that various specific combinations of API methods work as expected; they do not perform an exhaustive check that all combinations of API methods will work as expected. However, they do statically prove that the given tested method call sequences work for *all combinations of parameters*. For example, the last example in Fig. 5 establishes that `F.finverse` is the inverse of `F.f` for all values of the `int` parameter `i`. Dynamic testing can only establish this fact for a selection of values of `i`.

7 Determinism

When attempting to verify the implementation of a method that contains calls to other methods, the behavior of the called methods is known only by their specifications. If the called methods are underspecified, then questions of determinism can arise. To be concrete, suppose we are verifying method `m`, which repeatedly calls method `mm`, both in `m`'s implementation and its specification. Futher, suppose `mm` is *underspecified*, that is, that it's result is constrained but allows more than one result value. For example, the postcondition might simply be `ensures \result >= 0;` . The logical encoding of the calls to `mm` introduce, at each call site, a logical variable representing the result of the method call and constraints corresponding to the postconditions. There is then no constraint that the two successive calls return the same value. Determinism can be controlled by the specification technique described next.

```
class X {
  //@ model public int _length;

  //@ public normal_behavior
  //@   ensures \result >= 0 && \result == _length;
  public int length() { ... }

  //@ assignable _length;
  public void add(...) { ...}
}
```

Figure 6. Using a model variable to force determinism

Keep in mind three different cases:

- Two calls happen in the same program state. Then the same value is expected unless the method is *volatile* (cf. §8).
- Two calls happen in different program states but nothing has changed on which the result depends. This requires a specification of the information on which a method's result depends. Such information is given by `\from` or `reads` clauses, which are not commonly used and not currently implemented in OpenJML.
- Two calls happen in different program states between which something has changed relevant to the result. In this case, the two results should not be expected to be necessarily the same.

The way to explicitly control when a method's result remains fixed and when it may change is to introduce an uninterpreted model variable, as shown in Fig. 6. Here the value of `length()` is set equal to the value of the model variable `_length`. The value of `_length` and `length()` (for a given object) will always be the same, no matter what changes there are to the heap. A possible change is signaled by listing `_length` in an `assignable` clause, as shown in the (partial) specification for `add` in the Figure.

Some tools may implement a semantics and logical encoding that specifically guarantees determinism when there is no state change (in the absence of a `volatile` designation). JML and OpenJML may adopt this semantics in the future. However, even in that case, the idiom above is useful for detailed control of when method return values can be expected to change or not.

8 Volatility

The flip side to the problem of specifying determinism in the previous section is that of specifying volatile memory locations. In this context, *volatile* memory locations are those that may change without being assigned by the program being specified. Any value that can be modified externally is a candidate for volatility. We will take as an example the system clock. An initial simple specification is shown in Fig. 7. By using underspecification, each call to `currentTimeMillis` is allowed to return any value at all. There are two problems with this specification. First, we normally want to require

```
1 class System {
2   //@ public normal_behavior
3   //@   ensures true;
4   public static long currentTimeMillis() {
5 }
```

Figure 7. Inadequate system clock specification

```
1 class System {
2
3   //@ model public static long lastClockValue;
4   //@ model public static long nextClockValue;
5
6   //@ public normal_behavior
7   //@   assignable lastClockValue, nextClockValue;
8   //@   ensures lastClockValue == \old(nextClockValue);
9   //@   ensures \result == \old(nextClockValue);
10  //@   ensures nextClockValue >= \old(nextClockValue);
11  public static long currentTimeMillis();
12 }
```

Figure 8. Specifying the system clock

that the system clock only advance.[3] Second, it is useful to be able to refer to values of the system clock in specifications; without changing the system state.

Fig. 8 shows a more successful specification. It declares public model fields `lastClockValue` and `nextClockValue`, which can be directly referred to in specifications. The actual method itself returns the value of `nextClockValue`, which is specified to be non-decreasing. Upon a call of `currentTimeMillis`, the model fields shift values. The new value of `nextClockValue` is constrained only to be non-decreasing; listing it in the frame condition guarantees that it may change. This specification simulates the behavior of a real clock, though it does not have any connection to the passage of real time.

9 Loops

The state of the art in software verification using SMT solvers still requires manual specification of loop invariants, since SMT solvers do not do induction well. Automated inference of loop specifications is an active area of research that is not yet integrated into proof tools. However, many loops have a simple, common form that is amenable to an idiomatic specification.

Consider the loop in Fig. 9. It employs a common simple loop style: the loop variable is initialized and incremented by 1 to a fixed end. So the loop invariant on line 4 is easily stated; by using a common idiom one can avoid the error of writing `i < a.length`. The *decreases* clause on line 7 (used to prove termination) is also obvious in this simple loop. The loop frame condition (line 6) can be determined by inspection if the loop body is simple. This leaves just the inductive invariant on line 5. But since there is no interaction

[3]This may not always be the case; some systems may want to allow but guard against resetting of the clock to catch security violations.

```
1 class Tests {
2   public void m(int[] a) {
3     ...
4     //@ loop_invariant 0 <= i && i <= a.length;
5     //@ loop_invariant (\forall int j; 0<=j && j<i; a[j]==j);
6     //@ loop_modifies a[*], i;
7     //@ decreases a.length - i;
8     for (int i = 0; i < a.length; i++) {
9       a[i] = i;
10    }
11  }
12 }
```

Figure 9. Loop specifications

between the loop iterations, this too can be manually written by rote.

In more complicated scenarios, an approach to loop specifications can be taught:

- Identify a loop index that can be used to write a *decreases* clause and a loop invariant that constrains the loop index.
- Determine the set of memory locations that are changed in the loop; those make up the `loop_modifies` clause.
- Then write an inductive invariant for each such memory location; the invariant must specify all the values computed up to the current value of the loop index.

In fact, this formula is straightforward enough that it could be automated, providing loop inference, and avoiding the need to write loop specifications, at least for simple loops.

10 Abstraction

Students of programming are taught modularization and abstraction as means to organize complex systems. In Java, abstraction is often implemented by defining an *interface* that contains the public API to some capability and then one or more concrete classes that implement the functionality, but are not seen directly by the client. The Java system library contains many examples of this design pattern, such as `Collection`, `Map`, and `Comparator`. Fig. 10 shows a simple example of an interface that manages an abstraction of a single integer value. In this case, the implementation of the abstraction used in `ConcreteValue` is trivial and obvious (to conserve space), but could readily be something more complex.

The clients of this capability interact solely through the abstraction and see only the specifications of the abstraction. The question then is how to specify the abstraction without reference to the concrete instance. The idiom to use is model fields. The software engineer identifies the concepts that the abstraction is meant to represent and defines those within the Java interface as model fields. The specifications of the interface are then written in terms of those model fields. This is illustrated in Fig. 11. Note a few important points.

```
interface Value<T> {
  public T getValue();
  public T setValue(T v); // returns old value
}

class ConcreteValue<T> implements Value<T> {
   private T value;
   @Override public T getValue() { return value; }
   @Override public T setValue(T v) {
          T t = value; value = v; return t;
   }
}
```

Figure 10. Example of Abstraction

```
interface Value<T> {
  //@ model instance public T _value;
  //@ ensures \result == _value;
  public T getValue();
  //@ assignable _value;
  //@ ensures \result == \old(_value) && _value == v;
  public T setValue(T v); // returns old value
}

class ConcreteValue<T> implements Value<T> {
   private T value; //@ in _value;
   //@ private represents _value = value;
   @Override public T getValue() { return value; }
   @Override public T setValue(T v) {
          T t = value; value = v; return t;
   }
}
```

Figure 11. Abstraction

- The model field's value is defined by the assertions in the postconditions of the interface methods; it is not actually assigned anywhere.
- The methods of the concrete class inherit their specifications from the parent interface; no additional specifications are needed in the concrete class.
- The connection between the concrete field and the abstract model field is made by the `represents` clause. Here the representation is a simple equality, but could be an expression that indicates how the modeled integer is constructed from the concrete representation. The representation can also be stated using an implicit boolean constraint, using JML's `\such_that` syntax.
- The *assignable* clause in the interface states that `setValue` may change the model field; but `ConcreteValue.setValue` changes the *concrete* field `value`. So we need to state that the concrete field is included in the model field in the context of frame conditions. That is the purpose of the `in` clause attached to the declaration of `value`.

11 State machines

Sometimes it is necessary to control which methods of an API are permitted to be called when. That is, the specification needs to include a small state machine along with preconditions that indicate in which states particular methods can be called. Fig. 12 shows a simple example of a state machine specification using a ghost variable, which is assigned using JML `set` statements. Such a specification is also a good example of one that can benefit from including testing scenarios as described in §6.

```
class X  {
   //@ ghost public int state;

   //@ requires state == 0;
   //@ assignable state;
   //@ ensures state == 1;
   public void init() { /*@ set state = 1; */}

   //@ requires state == 1; // stays in state 1
   public void dosomething() { ... }

   //@ requires state == 1;
   //@ assignable state;
   //@ ensures state == 2;
   public void close() { ... /*@ set state = 2; */ }
}
```

Figure 12. Example of a state machine in a specification

12 Conclusion

The sections above have presented several common specification situations and illustrated specification idioms useful for those cases. Using a consistent style and approach to common situations aids correct writing and easy understanding. Such is particularly useful when the readers are new to formal specifications. In addition though, identified idioms are source material for researching opportunities to simplify specification languages by defining defaults, implementing specification inference for simple cases, or evolution of specification language semantics.

References

[1] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2003. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03) (Electronic Notes in Theoretical Computer Science (ENTCS))*, Thomas Arts and Wan Fokkink (Eds.), Vol. 80. Elsevier, 73–89. http://www.sciencedirect.com/science/journal/15710661

[2] David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi (Eds.). Lecture Notes in Computer Science, Vol. 6617. Springer Berlin Heidelberg, 472–479. https://doi.org/10.1007/978-3-642-20398-5_35

[3] David R. Cok. 2014. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In *Workshop on Formal Integrated Development Environment (F-IDE 2014) (EPTCS)*, Vol. 149. 79–92. https://doi.org/10.4204/EPTCS.149.8 Grenoble, France.

[4] David R. Cok. 2018. http://www.openjml.org.

[5] JML. 2018. Many papers regarding JML can be found on the JML web site: http://www.jmlspecs.org.

[6] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.