Writing JML Specifications Using Java 8 Streams

Yoonsik Cheon, Zejing Cao and Khandoker Rahad

TR #16-83
November 2016

Keywords: Assertions, formal specifications, lambda notations, streams, Java, JML.

1998 CR Categories: D.1.5 [Programming Techniques] Applicative (Functional) Programming; D.2.1 [Software Engineering] Requirements/Specifications—languages; D.2.4 [Software Engineering] Software/Program Verification—class invariants, formal methods, programming by contract; D.3.3 [*Software Engineering*] Language Constructs and Features—classes and objects, frameworks; 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

Writing JML Specifications Using Java 8 Streams

Yoonsik Cheon, Zejing Cao and Khandoker Rahad
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
ycheon@utep.edu, {zcao2, karahad}@miners.utep.edu

Abstract—JML is a formal behavioral interface specification language for Java to document Java program modules such as classes and interfaces. When composing JML specifications, one frequently writes assertions involving a collection of values. In this paper we propose to use Java 8 streams for writing more concise and cleaner assertions on a collection. The use of streams in JML can be minimal and non-invasive in the conventional style of writing assertions. It can also be holistic to write all assertions in the abstract state defined by streams. We perform a small case study to illustrate our approach and show its effectiveness as well. We then summarize our findings and the lessons that we learned from the case study.

Keywords—assertions, formal specifications, lambda notation, stream, Java, JML.

I. Introduction

Java 8 supports functional-style programming by introducing lambda expressions and a stream application program interface (API) [11]. A lambda expression is a block of code with parameters that can be passed around so that it can be executed later. A stream is an immutable sequence of elements, providing a variety of so-called higher-order operations such as *filter*, *map*, and *reduce* that take lambda expressions as arguments. The underlying idea is to convert a collection to a stream, process the elements potentially in parallel, and then gather the results into a collection. Elements are processed by pipelining stream operations. One key benefit of using streams is the internalization of iterations. The code is completely unaware of the iteration logic in the background.

The Java Modeling Language (JML) is a behavioral interface specification language for Java to formally specify the behavior of Java classes and interfaces [7] [1]. In JML, the behavior of a Java class is specified by writing class invariants and pre and post-conditions for the methods exported by the class. In designing an object-oriented program, a class relationship called an association plays an important role. It defines the internal structure of an object. An object is associated with other objects and collaborate with them to perform a task collectively by sending messages. In writing JML specifications, thus, it is crucial to manipulate a collection of objects effectively. In JML, one can write assertions on a collection of objects using quantified expressions like \forall and \exists. The JML quantifiers, however, are similar to external iterations in that one has to use quantified variables to iterate over a collection of objects. The current JML doesn't provide a notation for internal iteration over collections.

In this paper we propose to use the Java Stream API in JML. The aim is to write more concise and cleaner assertions at a higher level of abstraction. We explore several different ways of using streams in writing JML specifications. However, the underlying idea is the same and is to convert a collection to a stream and write assertions using various stream operations. The conversion can be done either explicitly or implicitly by defining an abstraction function. An abstraction function specifies a mapping from concrete program states to abstract specification states. The style of writing assertions can be minimalistic and non-invasive. One can mix stream assertions with those written in the conventional style. The assertion style can also be holistic in that one writes all assertions in terms of abstract streams, not concrete collections. For this, one uses *model fields*, specification-only fields introduced for writing JML specifications functions [4]. We explain our approach by applying it to a Battleship game application, a well-known guessing game for two players (see Section III). We show a series of example JML assertions to illustrate many interesting aspects of using streams in JML. We also point out interesting technical questions and future research directions to better support the use of streams in JML.

In Section II below we provide a quick overview of JML and Java 8 Stream API. In Section III we describe the Battleship game briefly along with its design expressed in a class diagram. In Section IV we illustrate our approach by writing many JML assertions involving the many ends of 1-to-many associations in the Battleship application. In Section V we describe some of the lessons that we learned along with possible improvements or extensions to JML to help the use of streams. In Section VI we provide a concluding remark.

II. Background

A. JML

The Java Modeling Languge (JML) is a behavioral interface specification language for Java to formally specify the behavior of Java classes and interfaces [7]. JML provides a wide range of tools from static analysis to runtime checking and interactive

verification [1]. In JML, the behavior of a Java class is specified by writing class invariants and pre and postconditions for the methods exported by the class. Listing 1 shows an example JML specification concerned with a game played by two players. As shown, JML specifications are written as special comments in Java source code, either after //@ or between /*@ and @*/. The keyword *spec_public* indicates that private fields *players* and *active* are treated as public for a specification purpose. They can be used in publicly-visible specifications such as public class invariants. One unique feature of JML compared with other specification languages like Z and VDM-SL is that JML assertions are written in the Java expression syntax with a few JML-specific extensions like universal and existential quantifiers. The first invariant constrains the length of the *players* array to 2 and the value of the *active* field to be a legal index of *players*. The next two invariants assert that the elements of *players* are distinct and not a null value. A JML-specific operator ==> denotes logical implication. A method specification precedes the declaration of the specified method. The *requires* clause specifies the precondition, the *assignable* clause specifies the frame condition, and the *ensures* clause specifies the postcondition. The keyword *\old* in a postcondition denotes the pre-state value of an expression. It is most commonly used in the specification of a mutation method such as the *changeTurn*() method that changes the state of an object.

Listing 1. Example JML specification

```
public class Game {
  private /*@ spec_public @*/ Player[] players;
  private /*@ spec_public @*/ int active;

  /*@ public invariant players.length == 2 &&
    @   0 <= active && active < players.length; @*/

 /*@ public invariant (\forall int i; 0 <= i && i < players.length;
    @   players[i] != null);  @*/

  /*@ public invariant (\forall int i, j; 0 <= i && i < players.length
    @    && 0 <= j && j < players.length;
    @    i != j ==> players[i] != player[j]);  @*/

  /*@ requires true;
    @ assignable active;
    @ ensures active != \old(active);
    @ ensures_redundantly
    @    active == \old(active + 1) % players.length); @*/
  public void changeTurn() { ... }
}
```

B. Java 8 Streams

Java 8 enables functional-style programming by providing lambda expressions and a stream API [11]. A lambda expression is a block of code with parameters that can be passed around so that it can be executed later. A stream is an immutable sequence of elements, providing a variety of so-called higher-order operations that take lambda expressions as arguments. The stream API allows one to work with a sequence of elements possibly in parallel without worrying about how the elements are stored or accessed. To perform a computation, stream operations are composed into a stream pipeline. A stream pipeline consists of a source, zero or more intermediate operations and a terminal operation. Streams are most often lazy in that computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed. A few key features of Java Stream API include:

- *Functional-style operations*: A variety of higher-order operations are provided, including *filter*, *map*, and *reduce* (also called *fold* in functional languages).
- *Lazy construction*: A stream is constructed lazily in that its elements are computed when a user demands it. This is contrary to a collection whose elements are computed before they become parts of the collection. A collection is constructed eagerly.
- *Concurrency*: Many parallel operations are provided to process the elements contained in a stream, while completely abstracting out the low level multithreading logic.
- *Pipeline:* The API is based on the idea of converting a collection to a stream, processing the elements possibly in parallel, and then gathering the results into a collection. The elements are processed by pipelining stream operations, zero or more so-called *intermediate* operations like *map* followed by a *termination* operation like *reduce* [11].

One key benefit of using streams is the internalization of iterations, called *internal iterations*. A conventional way to iterate through an array or collection is to use *for* loops or *iterators*, as shown below.

```
for (int i; i < players.length; i++) {
   players[i].setFleet(defaultFleet());
}
```

This iteration is called an *external iteration*, and the iteration is clearly visible in the code. The Stream API provides methods like *forEach* to internalize iterations (see below), and the code is completely unaware of the iteration logic in the background.

```
Stream.of(players).forEach(p -> p.setFleet(defaultFleet())
```

The static method *Stream.of* creates a new stream from an array, and the *forEach* operation performs a specified task on each element. There are many operations provided by the Stream API. Table 1 shows several immutable operations that are most useful in writing JML assertions.

Table 1. Stream API

Operation	Description
allMatch(*pred*)	All elements satisfy *pred*?
anyMatch(*pred*)	Any element satisfy *pred*?
filter(*pred*)	Select all elements satisfying *pred*

map(*mapper*)	Apply *mapper* to each element
reduce(*ident*,*accum*)	Combine (fold) all elements
distinct()	Select all distinct elements
findAny()	Pick an arbitrary element
collect(*collector*)	Convert to a collection

III. BATTLESHIP GAME

In the next section we will specify in JML a Java program that allows a user to play Battleship games; we wrote both a Java application and an Android app [2]. Battleship is a very well-known guessing game for two players, and its purpose is to sink all the ships of the opponent. The game is played on grids, usually 10×10, of squares. Each player has a fleet of ships, and each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically. Once the ships are secretly positioned on the grids of the players, the game proceeds in a series of rounds. In each round, each player takes a turn to make a shot to a square in the opponent's grid. A shot is either a 'hit' on a ship or a 'miss'. When all the squares of a ship have been hit, the ship sinks. If all of a player's ships have been sunk, the game is over and the opponent wins.

We will write JML specifications of the classes found in the business logic layer. These classes are independent of a particular UI framework such as Java Swing and Android. Figure 1 describes the main business logic classes of the program and their relationships. A game consists of two players, each with a board and a fleet of ships. The ships of a player are to be placed on the player's board by the player and then to be hit and sunk by the opponent player. Our JML specifications will be focused on the 1-to-many associations.

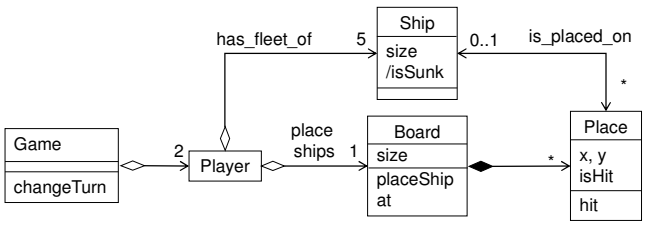

Figure 1. Battleship class diagram

IV. USING STREAMS IN JML

In this section we will suggest several different ways for writing better JML specifications using streams. As shown in the class diagram in the previous section, there are four 1-to-many associations in the Battleship application. The many ends of these associations will be represented as either collections or arrays. We will write JML assertions manipulating the many ends of these associations using streams.

A. Writing Assertion

The simplest way of using streams in JML is to convert an array or collection to a stream inside an assertion and write the assertion in terms of the stream. As examples, consider the following two invariants from Listing 1 in Section II.A.

```
I1: (\forall int i; 0 <= i && i < players.length; players[i] != null);
I2: (\forall int i, j; 0 <= i && i < players.length;
   0 <= j && j < players.length;
   i != j ==> players[i] != player[j]);
```

The first invariant states that the *players* array shouldn't contain a null value, and the second states that there is no duplicate player contained in the array. Both invariants use the JML universal quantifier, and thus they are in a sense external iterations. We can internalize the iterations or simplify the assertions using stream operations as follows.

```
I1: Stream.of(players).matchAll(p -> p != null);
I2: Stream.of(players).distinct().count() == players.length;
```

The *Stream.of* static method creates a stream from an array. The *matchAll* method tests if each element of a stream satisfies the specified condition written in lambda notation. The second invariant (I2) asserts that the number of distinct elements of the stream is the same as the length of the *players* array. The *distinct* method creates a new stream by collecting all distinct elements of a stream. The use of stream methods like *matchAll* produces assertions that are concise and easy to read and understand. This is due to the internalization of iterations. The assertions are cleaner, as there is no need to introduce quantified variables and manipulate them to access the elements explicitly. The invariant I2 demonstrates that assertions can be written at a higher abstraction level with stream operations. It constrains the size of a stream instead of comparing each pair of the elements. An equivalent assertion written with *players* would be: *new HashSet<Player>(players).size() == players.length*.

This simple approach works well if stream operations are used sparsely in assertions. As shown in I2, stream expressions can also be mixed with other expressions like *players.length*. However, one weakness of this simple approach is duplications of conversion expressions like *Stream.of(players)* because of explicit conversions from arrays and collections to streams. Duplicates are bad in code and specifications as well. Another weakness is that a single assertion is written at two different levels of abstraction. A stream is an abstraction of an array or a collection. Thus, a stream expression is in a sense an abstract assertion written in terms of an abstract state. The problem is that abstract assertions are mingled with concrete assertions written in terms of concrete representations like arrays and collections. Such assertions are in general hard to read and understand because of constant shifts of abstraction levels.

B. Using Model Variables

A more holistic approach would be to write all JML assertions in terms of streams rather than underlying concrete representations (arrays or collections). For this we use JML *model fields*, specification-only fields introduced for writing JML specifications [4]. The use of model fields also makes the conversion from collections to streams occur implicitly. The key to this approach is to:

- Declare model fields of stream types
- Define abstraction functions for model fields
- Write abstract assertions by referring to model fields

As an example, let's rewrite the specification of the Game class. First, we define model fields.

```
//@ public model Stream<Player> specPlayers;
//@ public model activePlayer;
```

Note that we also abstract the notion of active player from its concrete representation, the index of the active player, to the player herself. We then define abstraction functions for model fields by mapping concrete values such as arrays and collections to abstract values of streams. The abstraction functions allow one to evaluate abstract assertions written with model fields in concrete program states [4]. JML provides *represents* clauses to define abstraction functions.

```
private Player[] players; //@ in specPlayers;
private int active; //@ in activePlayer;
//@ private represents specPlayers <- Stream.of(players);
//@ private represents activePlayer <- players[active];
```

As shown above, the abstraction functions for both model fields are private and specified straightforwardly. The *in* clause following a field like *players* adds the field to a data group. A *data group* is a set of locations and is used in JML's frame axioms (*assignable* clauses) to name sets of locations in a way that does not expose representation details [8] (see below for an example). Each model fields define a new data group of the same name.

Now we can write JML assertions such as class invariants and method pre and postconditions by referring to only model fields. For example, we can rewrite all the invariants of the Game class as follows.

```
/*@ public invariant specPlayers.count() == 2 &&
  @    specPlayers.distinct().count() == 2 &&
  @    specPlayers.allMatch(p -> p != null) ; @*/

//@ public invariant specPlayers.anyMatch(p -> p == activePlayer);
```

As expected, there is no explicit conversion from arrays to streams, and all assertions concerned with players are at the same abstraction level. They are all written in terms of *specPlayers*. In the public scope, game players are viewed as a stream of players, not an array. The assertions are shorter and more readable. We can also rewrite the specification of the *changeTurn* method. As hinted earlier, the *assignable* clause also uses a model field—to be more precise, the data group of the model field—not the concrete representation.

```
/*@ assignable activePlayer;
  @ ensures activePlayer != \old(activePlayer);
  @ ensures_redundantly activePlayer ==
  @   specPlayers.filter(p -> p != activePlayer).findAny().get(); @*/
public void changeTurn() { … }
```

There is another important benefit of using model fields. Recall from the model field declarations that concrete representations such as *players* and *active* remain private, and *represents* clauses are also private. Therefore, they may be changed without affecting public assertions and any assertions written in terms of public model fields such as *specPlayers* and *activePlayer*. It is also possible to keep all previous assertions as private for the implementer of the Game class.

C. Streams to Collections

The Stream API defines a variety of higher-order operations such as *allMatch*, *anyMatch*, *filter*, *map*, and *reduce*. These operations provide a convenient way to iterate through the elements of a stream. When writing JML assertions, however, there are cases in which it would be preferable to manipulate the elements as a collection by applying collection operations. As an example, consider the *tail*() method of the Ship class. A ship occupies a sequence of consecutive places in a board, and the method returns the last place of the sequence.

```
/*@ requires specPlaces.count() > 0;
  @ ensures \result == specPlaces.skip(specPlaces.count()-1)
  @    .findFirst().get();
  public /*@ pure @*/ Place tail() { ... }
```

The pre and postconditions are written using a model field *specPlaces* that represents a sequence of places occupied by a ship. The postcondition is convoluted because there is no stream operation to access an element based on its position. In fact, such an operation would defeat the purpose of a stream. If we use the concrete representation, a private field *places* of type List, the postcondition can be simplified to:

```
//@ ensures \result == places.get(places.size() – 1);
```

However, it not only exposes the implementation details but also results in a mixed use of abstract and concrete values. What is missing is a unified abstraction that supports both the stream and collection operations (refer to Section V for a discussion). A quick solution would be to map streams to appropriate collections. The collections could be either Java collections such as Set and List or JML collections such as JMLEqualsSet and JMLEqualsSequence. JML provides a set of immutable collection types suitable for writing assertions. For example, the above postcondition can be rewritten to:

```
/*@ ensures \result == specPlaces.collect(Collectors.toList())
  @    .get(specPlaces.count() – 1); @*/
```

The *Collectors.toList*() collector accumulates the elements of a stream into a new list. If the conversion is needed at several places, we may define a model method for that. Like a model field, a *model method* is a method defined solely for writing JML specifications [4]. A model method would be especially usual for converting a stream to a JML collection class or defining a JML-specific collector.

```
/*@ public pure <T> JMLEqualsSet<T> toSet(Stream<T> stream) {
  @    return JMLEqualsSet.convertFrom(
  @      stream.collect(Collectors.toList())); @*/
```

Note that converting a stream to a collection is different from using a concrete collection representation as done in Section A. It doesn't expose the implementation details and isn't limited to a particular collection type.

D. More Examples

In this section we show a serious of JML specifications to further illustrate the use of streams in writing assertions. All the examples are from the Battleship application. The complete specifications of the Battleship classes can be found in Appendix.

The *Player* class is an abstraction of a Battleship game player, and each player has a board and a fleet of ships. A player's fleet of ships is abstracted to a model field named *specFleet* as shown below.

```
//@ public model Stream<Ship> specFleet;
private /*@ spec_public @*/ Board board;
private List<Ship> fleet; //@ in specFleet;
//@ private represents specFleet <- fleet.stream();
```

One interesting domain constraint is that a player has at least one ship of size from 2 to 5, inclusive. Another constraint is that all the ships placed on the board of a player belong to the player. A board keeps track of all the ships placed on it (see the specification of the Board class below). These two constraints can be expressed as invariants as follows.

```
/*@ public invariant IntStream.rangeClosed(2,5).allMatch(n ->
  @   specFleet.anyMatch(ship -> ship.size() == n));
  @ public invariant
  @   toSet(specFleet).containsAll(toSet(board.specShips)); @*/
```

The first invariant uses a stream of integers from 2 to 5, inclusive. Its use produces a concise assertion by eliminating the use of nested quantifiers. Compare it with the following assertion written without a stream.

```
(\forall s: int; 2 <= s && s <= 5;
  (\exists i: int; 0 <= i && i < fleet.size();
    fleet.get(i).size() == s));
```

The second constraint asserts a subset relationship between two sets of ships. It can also be expressed with streams, but using a set operation like *containsAll* produces a more concise assertion. Thus, we convert streams to sets using a model method *toSet* (see Appendix for the definition).

A player's fleet of ships can be obtained by calling the *fleet* method, which returns an *Iterable*. The behavior of this method can be specified nicely using a stream. The stream obtained from the returned iterable object should be equivalent to *specFleet*. The *StreamSupport.stream* static method creates a new stream from an iterable object.

```
/*@ ensures specFleet.equals(
  @    StreamSupport.stream(\result.spliterator(), false)) @*/
public /*@ pure @*/ Iterable<Ship> fleet() { ... }
```

The *Ship* class is an abstraction of a battleship that can be placed on a player's board and then hit and sunk by the opposing player. Each ship has a size and a sequence of places. The abstract and concrete states of a ship are represented as follows.

```
//@ public model Stream<Place> specPlaces;
private /*@ spec_public @*/ final int size;
private final List<Place> places; //@ in specPlaces;
//@ private represents specPlace <- places.stream().sorted(cmp);
```

The abstract state of a ship's places, *specPlaces*, is interesting in that it is a sorted stream. The comparator *cmp* used in the *represents* clause is a final model field that compares two places considering only their column and row indices (see Appendix for the definition). A sorted stream facilitates writing certain assertions, e.g., the postcondition of the *head* method shown below.

```
/*@ requires specPlaces.count() > 0;
  @ ensures \result == specPlaces.findFirst(); @*//
public /*@ pure @*/ Place head() { ... }
```

One key domain constraint is that a ship occupies a sequence of consecutive places, and the number of places should be less than or equal to the size of the ship. The number of places may be less than the size of the ship because the ship may be in the process of being placed one place at a time. The first constraint may be specified directly in terms of the concrete representation (field *places*) as follows.

```
places.size() == size ==>
  (\forall int i; 0 < i && i < places.size();
    places.get(i).getX() == places.get(i-1).getX() + 1
    && places.get(i).getY() == places.get(i-1).getY()) ||
  (\forall int i; 0 < i && i < places.size();
    places.get(i).getX() == places.get(i-1).getX()
    && places.get(i).getY() == places.get(i-1).getY() + 1);
```

Besides readability, one downside of the above assertion is that it relies on the underlying implementation details. It assumes that the places are stored in increasing order of their column or row indices. The use of model fields allows us to make such an assumption safely at the abstract state. In fact, our abstraction function already maps a list of places to a sorted stream. We can also remedy the problem by reformulating the assertion to manipulate streams of column and row indices. We can use such operations as *max* and *min* defined for IntStream.

For this, we first convert a stream of places to a stream of row or column indices using the *maptToInt* method shown below in the *where* clause. The *where* clause is our own extension to JML to introduce local definitions.

```
specFleet.count() == size ==>
(horizontal && xs.max().getAsInt() == xs.min().getAsInt() + size - 1)
||| (vertical && ys.max().getAsInt() == xs.min().getAsInt() + size - 1)
where
  IntStream xs = specPlaces.mapToInt(p -> p.getX()).distinct();
  IntStream ys = specPlaces.mapToInt(p -> p.getY()).distinct();
  boolean horizontal = ys.count() == 1 && xs.count() == size;
  boolean vertical = xs.count() == 1 && ys.count() == size;
```

Perhaps, the most interesting class of the Battleship application is the *Board* class, an abstraction of a battleship board. A board is composed of $n \times n$ places, where $n > 0$, and has a set of ships placed on it. The state of a board is represented as follows.

```
//@ public model Stream<Place> specPlaces;
//@ public model Stream<Ship> specShips;
private /*@ spec_public @*/ final int size;
private final List<Place> places; //@ in specPlaces;
private final List<Ship> ships; //@ in specShips;
//@ private represents specPlaces <- places.stream();
//@ private represents specShips <- ships.stream();
```

There are many interesting constraints on a board (see Appendix). One such a constraint is the uniqueness of the indices of the places belonging to a board. A pair of column and row indices, (x, y), should uniquely identify a place of a board, where $0 \leq x, y < size$. This can be specified in terms of concrete representation (*places*) using nested quantifiers as follows.

```
(\forall int i; 0 <= i && i < places.size();
  (\forall int j; 0 <= j && j < places.size();
    i != j ==> places.get(i).getX() != places.get(j).getX()
      || places.get(i).getY() != places.get(j).getY()))
```

By using a stream, we can simplify the above assertion as follows.

```
specPlaces.count() ==
  specPlaces.map(p -> p.getX() + "," + p.getY()).distinct().count()
```

It asserts the uniqueness of indices indirectly by constraining the number of column-and-row indices. The use of string concatenation is a quick workaround to represent a pair of values for a counting purpose. One may introduce a model type to represents a pair of values or use a JML model class such as JMLValueValuePair to represent a pair of values.

Another interesting constraint is that all the ships of a board are indeed placed on the board. We can specify this constraint in terms of concrete representations (*ships* and *places*) or their abstractions (*specShips* and *specPlaces*). The *flatMap* operation used in the second assertion flattens the results, e.g., transforms a stream of streams to a stream of elements.

```
(\forall int i; 0 <= i && i < ships.size();
    (\forall int j; 0 <= j && j < ships.get(i).places.size();
      places.contains(ships.get(i).places().get(j))));

specShips.flatMap(s -> s.specPlaces)
     .allMatch(p -> toSet(specPlaces).contains(p));
```

Obviously, two ships cannot overlap. Stating this directly in terms of the concrete representation (*ships*) is a bit involved. It requires nesting of several quantifiers.

```
(\forall int i, j;
   0 <= i && i < ships.size() && 0 <= j && j < ships.size();
   i != j ==>
     (\forall int i1; 0 <= i1 && i1 < ships.get(i).places.size();
       (\forall int j1; 0 <= j1 && j1 < ships.get(j).places.size();
         ships.get(i).places().get(i1) !=  ships.get(j).places().get(j1))));
```

It can be stated indirectly using streams by constraining the number of places. The number of places of all ships should be equal to the number of unique indices of the places.

```
all.count() == all.map(p -> p.getX() +","+ p.getY()).distinct().count()
where
  Stream<Place> all = specShips.flatMap(s -> s.specPlaces);
```

So far, we focused on specifying class invariants. Streams can be used equally well in specifying the behavior of a method and a constructor. They provide a powerful way of writing assertions involving collections in a single state, such as class invariants, method preconditions, and postconditions of observer methods. For example, the following two observer methods of the *Board* class can be nicely specified using streams. In particular, the use of the *orElse* method on an *Optional* object emphasizes the fact that the *at* method may return a null value. An optional object is a container, and the *orElse* method returns the contained value or the specified value if there is no value present.

```
/*@ ensures \result == specPlaces.filter(p ->
  @   p.getX() == x && p.getY() == y).findAny().orElse(null); @*/
public /*@ pure nullable @*/ Place at(int x, int y) { ... }

//@ ensures \result == specShips.allMatch(s -> s.isSunk());
public /*@ pure @*/ boolean allSunk() { ... }
```

However, streams often aren't effective in specifying state changes of mutation methods. Their assertions involve two states, pre- and post-states. To specify the side effect of a method, one needs to relate the new value of a stream in the post-state to its old value in the pre-state. But, there are not many operations provided for comparing or relating two streams. As an example, consider the *placeShip* method of the *Board* class that, given a start place and a horizontal or vertical direction, places a ship on a board. Its specification is shown below.

```
/*@ requires (* omitted *);
  @ assignable specShips, specPlaces, ship.specPlaces;
  @ ensures specShips.equals(
```

```
  @    \old(Stream.concat(specShips, Stream.of(ship))));
  @ ensures specPlaces.equals(\old(specPlaces)) &&
  @    specPlaces->allMatch(p ->
  @       p.ship == (pls.constains(p) ? ship : \old(p.ship)));
  @ ensures toSet(ship.specPlaces).equals(pls);

  @ where Set<Place> pls = specPlaces.filter(p ->
  @   dir ? x <= p.x && p.x < x + ship.size() && p.y == y
  @     : p.x == x && y <= p.y && p.y < y + ship.size())
  @   .collect(Collectors.toSet()); @*/
public void placeShip(Ship ship, int x, int y, boolean dir) { ... }
```

The above specification uses three different ways of relating two streams: comparing two streams directly (the first *ensures* clause), stating properties using stream methods (the second *ensures* clause), and converting to collections such as sets (the third *ensures* clauses). In many cases, the third approach is most effective due to the availability of a large number of collection types and their operations (including JML model classes).

V. Lessons Learned and Discussion

In this section we describe some of the lessons that we learned from specifying the Battleship application. We also suggest a few possible extensions to JML to help effective use of streams.

The use of streams in assertions allows us to pick a suitable style for writing assertions. In particular, they enable us to write constructive assertions for method postconditions. As an example, let's reconsider the *at* method of the *Board* class specified in the previous section. The method returns a place at the given indices.

```
/*@ ensures \result == specPlaces.filter(p ->
  @   p.getX() == x && p.getY() == y).findAny().orElse(null); @*/
public /*@ pure nullable @*/ Place at(int x, int y) { ... }
```

If we specify the method without using a stream, we will have something similar to the following.

```
/*@  requires 0 <= x && x < size && 0 <= y && y < size;
  @  ensures places.contains(\result) &&
  @    \result.getX() == x && \result.getY() == y;
  @ also
  @  requires x < 0 || x >= size || y < 0 || y >= size;
  @  ensures \result = null; @*/
```

The *also* keyword separates different cases of a specification, and the method has to satisfy all the specification cases. Compare this specification with the stream version above. The stream version is constructive in that it states how the result is calculated while this one is property-oriented in that it only states the property that the result has to meet. A constructive assertion is often more intuitive and understandable. It also provides a guidance to an implementer. With streams, one can pick an assertion style that works best for a particular situation: constructive, property-oriented, or a combination of both.

We found that writing assertions at a higher level of abstraction using streams help us to expose weaknesses in our assertions. In Section II.A, we asserted that two players of a game should be different by writing the following invariant.

```
(\forall int i, j;
  0 <= i && i < players.length && 0 <= j && j < players.length;
  i != j ==> players[i] != player[j])
```

The invariant looked good. But, when we wrote a stream version, *specPlayers.distinct*().*count*() == *specPlayers.count*(), it stoke us that the *distinct* method uses the *equals* method to identify all distinct elements of a stream. We then realized that the original invariant is too weak in that it uses an object equality (==) to compare players. The array shouldn't contain more than one equivalent player by using the *equals* method. Similarly, the use of streams also let us uncover several important implicit assumptions present in the code. For all the 1-to-many associations of the application, the code made an implicit assumption that the many ends contain no duplicates. This is the default in the class diagram because the many ends have a uniqueness property by default. But, when the associations are translated to Java arrays or collections like lists, it has to be stated explicitly as an invariant. Our initial JML specification missed them. We often needed to introduce our own, customized notion of duplication without relying on the *equals* method of the elements. We were able to formulate it easily by using streams. For example, all places of a ship have to be unique on their column and row indices, as specified below.

```
specPlaces.map(p -> p.getX() + "," + p.getY()).distinct().count()
  == specPlaces.count()
```

We learned that JML is good in identifying and writing assertions for individual program modules such as classes. However, we often missed important, high-level domain constraints buried in the code, especially those involving multiple modules. One such a constraint is that the numbers of ships and their sizes should be the same for both players of the game. It's so basic and fundamental that we didn't even think about it or include an invariant for it in the *Game* class.

```
specPlayers..allMatch(p1 ->
   specPlayers.allMatch(p2 ->
     p1.specFleet.mapToInt(s -> s.size()).sorted().equals(
        p2.specFleet.mapToInt(s -> s.size()).sorted())
```

We identified the missing invariant while studying our UML class diagram and formulating some of the domain constraints in OCL. The Object Constraint Language (OCL) is a textual notation to specify constraints on UML models that cannot otherwise be expressed using diagrammatic notations such as class diagrams [11]. We believe that such inter-module constraints can be identified better using a design notation such as the UML class diagram that shows an overall structure of an application. It is also said assertions are more effective when derived from formal specifications such as OCL constraints [11] [9]. It would be possible to systematically translate OCL constraints to JML assertions [5].

As hinted in the previous section, streams are not a silver bullet. One motivation for the introduction of streams in Java 8 was to allow parallel or concurrent operations [11]. This means that stream operations must be independent of the position of the elements in the stream or the elements around it. Thus, they make certain position-based assertions more complex. For example, to denote the *i*-th element of a stream, one use an expression like: *stream.skip*(*i-1*).*findFirst*().*get*(). For a similar reason, streams lack high-level collection and sequence operators commonly found in formal specification languages such as Z [10], VDM-SL [6], and OCL [11], and thus they are not good for asserting state changes of mutation methods. Our quick solution to the above problem was to convert a stream to a suitable collection on-the-fly. For this, you introduced utility model methods such as *toList* and *toSet*. However, a better solution would be to define sort of specification purpose, unified collection types to support common collection operations as well as stream operations that take lambda expressions. The idea is to define a set of JML model types similar to the OCL collection types that provide so-called *collection iterators* [11]. The various JML collection types could be a starting point for defining such unified collection types.

One key benefit of using streams is the internalization of iterations (see Section II.B), which becomes possible due to the introduction of lambda expressions in Java. Many stream operations take lambda expressions as arguments. In this paper we used only a very simple form of lambda expressions, those consisting of a single expression specifying the return value. We also used the lambda notation liberally without concerning much about technical details. However, there are many interesting technical questions regarding the use of lambda expressions in JML assertions. What kinds of statements are allowed in the body of a lambda expression? Can a model field be used in the body? Should the body be side-effect free? If so, how can it be assured? Can a lambda expression have its own specification? If so, can its body, a block of Java code, be completely left out? It would be interesting future research to fresh out these and other technical details and study the implications of using the lambda notation in JML.

A recommended pattern for using streams are: (a) convert a collection to a stream, (b) perform a series of stream operations such as filtering and mapping, optionally followed by reduction, and (c) convert the result stream back to a collection [11]. The main step frequently involves mutating the items contained in the stream using such stream operations as *forEach*. However, these stream operations shouldn't be used in JML assertions because of their side-effects. This makes it very difficult to express side-effects on streams in JML assertions. It is particularly problematic to map a stream of items to another stream by changing only parts of the states of the items. For example, the specification of the *at* method of the *Board* class shown in the previous section is incomplete. Only relevant parts are copied below.

```
/*@ assignable specPlaces;
  @ ensures specPlaces.equals(\old(specPlaces)) &&
  @    specPlaces->allMatch(p ->
  @       p.ship == (pls.constains(p) ? ship : \old(p.ship));
  @ where Set<Place> pls = …
  @*/
public void placeShip(Ship ship, int x, int y, boolean dir) { ... }
```

The intention of the *ensures* clause is to assert that for each place contained in *pls* its *ship* field should be set to *ship*; all other places should remain the same. But, the assertion is too weak in that it doesn't constrain other fields of *p* except for the *ship* field; thus, they may have any arbitrary values. One possible solution would be to improve the expressiveness of frame conditions to state exact locations that may be changed. We may introduce a regular expression notation or set-theoretic operations such as union, intersection, and complement to pinpoint a specific set of locations at a fine granularity. For example, the *\not_assigned* clause below states that all other fields of *p* except for the *ship* field are not allowed to be changed; that is, for the object *p*, only its *ship* field may be changed.

```
specPlaces.allMatch(p -> p.ship == ship && \not_assigned(p.!ship))
```

Another possibility would be to provide a built-in operation to denote a new state of an object by stating only those parts that are changed. The idea is to write an expression like: *p* with its *ship* field changed to *s* but all other fields remaining the same. We can borrow the \mu notation from Z [10] to state that as shown below, and we can express a mapping from one stream to another concisely.

```
specPlaces.equals(\old(specPlaces.map(p ->(\mu p; p.ship == ship))))
```

VI. Conclusion

We showed through a small case study how to use Java 8 streams in JML. The use of streams along with lambda expressions produces assertions that are more concise and cleaner. It also provides more options for selecting an appropriate assertion style: constructive, property-oriented, and a combination of both. However, streams are not a silver bullet. There are limitations on using them as they are, e.g., lack of a unified interface for both collection and stream operations, and thus ineffectiveness in asserting state changes or side-effects. There are also some technical details to fresh out for the full use of streams in JML. We suggested these and other interesting research questions as future work. One such a research question not discussed before is to study the impact of stream-based assertions on various JML tools, especially static checkers and verification tools.

References

[1] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. 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.

[2] Y. Cheon, *Are Java Programming Best Practices Also Best Practices for Android?* Technical Report 16-76, Department of Computer Science, The University of Texas at El Paso, El Paso, TX, October 2016.

[3] Y. Cheon, C. Avila, S. Roach, and C. Munoz, Checking design constraints at run-time using OCL and AspectJ, *International Journal of Software Engineering*, 2(3): 5–28, December 2009.

[4] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards, Model variables: Cleanly supporting abstraction in design by contract, *Software—Practice & Experience*, 35(6): 583–599, May 2005.
[5] A. Hamie, Using patterns to map OCL constraints to JML specifications, *International Conference on Model-Driven Engineering and Software Development*, pages 35-48, Lisbon, Portugal, January 2014.
[6] C. B. Jones, *Systematic Software Development using VDM*, Prentice Hall, 1990.
[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,* 31(3): 1–38, March 2006.
[8] K. R .M. Leino, Data group: Specifying the modification of extended state, *Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 144-153, ACM, 1998.
[9] D. S. Rosenblum, A practical approach to programming with assertions, *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
[10] J. M. Spivey, *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, New York, NY, 1988.
[11] R. Warburton, *Java 8 Lambdas: Functional Programming for the Masses*, O'Reilly, 2014.
[12] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, second edition. Addison-Wesley, 2003.

APPENDIX

This appendix provides specifications of all the classes of the Battleship program mentioned in this paper. Our specifications are not complete in that we only show several representative methods for each class.

A. Game

```
public class Game {
 //@ public model Stream<Player> specPlayers;
 /*@ public invariant specPlayers.count() == 2 &&
   @    specPlayers.distinct().count() == 2 &&
   @    specPlayers.allMatch(p -> p != null) &&
   @    specPlayers.allMatch(p1 -> specPlayers.allMatch(p2 ->
   @      p1.specFleet.mapToInt(s -> s.size()).sorted().equals(
   @          p2.specFleet.mapToInt(s -> s.size()).sorted()); @*/

 //@ public model Player activePlayer;
 //@ public invariant specPlayers.anyMatch(p -> p == activePlayer);

 private Player[] players; //@ in specPlayers;
 private int active; //@ in activePlayer;
 //@ private represents specPlayers <- Stream.of(players);
 //@ private represents activePlayer <- players[active];

 /*@ assignable activePlayer;
   @ ensures activePlayer != \old(activePlayer);
   @ ensures_redundantly activePlayer ==
   @   specPlayers.filter(p -> p != activePlayer).findAny().get(); @*/
 public void changeTurn() { … }
}
```

B. Player

```
public class Player {
 //@ public model Stream<Ship> specFleet;
 /*@ public invariant specFleet.count() >= 5 &&
   @    specFleet.distinct.count() == specFleet.count() &&
   @    IntStream.rangeClosed(2,5).allMatch(n ->
   @       specFleet.anyMatch(s -> s.size() == n)) &&
   @    toSet(specFleet).containsAll(toSet(board.specShips))); @*/

 private /*@ spec_public @*/ Board board;
 private List<Ship> fleet; //@ in specFleet;
 //@ private represents specFleet <- fleet.stream();

 /*@ requires specFleet.count() >= 5 &&
   @   specFleet.distinct.count() == specFleet.count() &&
   @   IntStream.rangeClosed(2,5).allMatch(n ->
   @      specFleet.anyMatch(s -> s.size() == n));
   @ requires board.specShips.count() == 0;
   @ assignable this.board, specFleet;
   @ ensures this.board = board && specFleet.equals(fleet.stream()); @*/
 public Player(Board board, List<Ship> fleet) { ... }

 /*@ ensures StreamSupport.stream(\result.spliterator(), false)
   @    .equals(specFleet); @*/
 public /*@ pure @*/ Iterable<Ship> fleet() { ... }

/*@ ensures \result == stream.collect(Collectors.toSet());
  @ public model pure <T> Set<T> toSet(Stream<T> stream) {
  @   return stream.collect(Collectors.toSet());
  @ } @*/
}
```

C. BOARD

```
public class Board {
 private /*@ spec_public @*/ final int size;
 //@ public model Stream<Place> specPlaces;
 //@ public model Stream<Ship> specShips;

 /*@ public invariant size >
   @   specShips.mapToInt(s->s.size()).max().orElse(0); @*/
 /*@ public invariant specPlaces.count() == size * size &&
   @    specPlaces.allMatch(p.getX() < size && p.getY() < size) &&
   @    specPlaces.count() ==
   @      specPlaces.map(p->p.getX() + "," + p.getY()).distinct().count(); @*/

 /*@ public invariant specShips.count() == specShips.distinct().count() &&
   @    all.count() == all.map(p -> p.getX() +","+ p.getY()).distinct().count()
   @    where Stream<Place> all = specShips.flatMap(s -> s.specPlaces); @*/

 private final List<Place> places; //@ in specPlaces;
 private final List<Ship> ships; //@ in specShips;
 //@ private represents specPlaces <- places.stream();
 //@ private represents specShips <- ships.stream();

 /*@ requires size > 0;
   @ assignable this.size, specPlaces, specShips;
   @ ensures this.size == size;
   @ ensures specShips.count() == 0;
   @ ensures specPlaces.allMatch(p -> p.isEmpty() && !p.isHit()); @*/
 public Board(int size) { ... }

 /*@ requires ship.specPlaces.count() == 0;
   @ requires specShips.noneMatch(p -> p.equals(ship));
   @ requires 0 <= x && x < size && 0 <= y && y < size;
   @ requires dir ==> x + len - 1 < size &&
   @   (\forall int i; x <= i && i < x + len; at(i,y).isEmpty());
   @ requires !dir ==> y + len - 1 < size &&
   @   (\forall int i; y <= i && i < y + len; at(x,i).isEmpty());
   @ assignable ship.specPlaces, specShips;
   @ ensures specShips.equals(
   @   \old(Stream.concat(specShips, Stream.of(ship))));
   @ ensures specPlaces.equals(\old(specPlaces)) &&
   @   specPlaces->allMatch(p ->
   @     p.ship == (pls.constains(p) ? ship : \old(p.ship)));
   @ ensures toSet(ship.specPlaces).equals(pls);
   @ where Set<Place> pls = specPlaces.filter(p ->
   @   dir ? x <= p.x && p.x < x + ship.size() && p.y == y
   @   : p.x == x && y <= p.y && p.y < y + ship.size())
   @   .collect(Collectors.toSet()); @*/
 public void placeShip(Ship ship, int x, int y, boolean dir) { ... }

 /*@ ensures \result == places.stream().filter(p ->
   @   p.getX() == x && p.getY() == y).findAny().orElse(null); @*/
 public /*@ pure @*/ Place at(int x, int y) { ... }

 //@ ensures \result == ships.stream().allMatch(s -> s.isSunk());
 public /*@ pure @*/ boolean isGameOver() { ... }
```

```
}
```

D. PLACE

```
public class Place {
  public /*@ spec_public @*/ final int x;
  public /*@ spec_public @*/ final int y;
  private /*@ spec_public @*/ boolean isHit;
  private /*@ spec_public nullable @*/ Ship ship;

  //@ public invariant x >= 0 && y >= 0;
  //@ public invariant ship != null ==> toSet(ship.specPlaces).contains(this);

  /*@ requires x >= 0 && y >= 0;
    @ assignable this.*;
    @ ensures this.x = x && this.y = y;
    @ ensures !isHit;
    @ ensures ship == null; @*/
  public Place(int x, int y) { ... }

  //@ ensures \result == x;
  public /*@ pure @*/ int getX() { ... }

  //@ ensures \result == y;
  public /*@ pure @*/ int getY() { ... }

  //@ ensures \result == isHit;
  public /*@ pure @*/ boolean isHit() { ... }

  /*@ assignable isHit;
    @ ensures isHit; @*/
  public void hit() { ... }

  //@ ensures \result == (ship != null);
  public /*@ pure @*/ boolean hasShip() { ... }

  //@ ensures result == (ship == null);
  public /*@ pure @*/ boolean isEmpty() { ... }

  /*@ requires isEmpty();
    @ requires ship.specPlaces.count() < ship.size() &&
    @    !toSet(ship.specPlaces).contains(this);
    @ requires ship.specPlaces.count() == ship.size() – 1
    @    ==> Ship.isSeq(specPlaces.concat(Stream.of(this)));
    @ assignable this.ship, ship.specPlaces;
    @ ensures this.ship == ship;
    @ ensures toSet(ship.specPlaces).equals(\old(toSet(
    @    specPlaces.concat(Stream.of(this))))); @*/
  public void placeShip(Ship ship) { ... }

  //@ ensures \result == ship;
  public /*@ pure @*/ Ship ship() { ... }

  /*@ public static model pure boolean isSeq(Stream<Place> stream) {
    @   int len = stream.count();
    @   IntStream xs = stream.mapToInt(p -> p.getX()).distinct();
    @   IntStream ys = stream.mapToInt(p -> p.getY()).distinct();
    @   return (xs.count() == len && ys.count() == 1 &&
    @       xs.max().getAsInt() == xs.min().getAsInt() + len – 1) ||
    @     (xs.count() == 1 && ys.count() == len &&
    @       ys.max().getAsInt() == ys.min().getAsInt() + len – 1);
    @ } @*/
}
```

E. SHIP

```
public class Ship {
  private /*@ spec_public @*/ final int size;
  //@ public invariant 2 <= size && size <= 5;

  /*@ public model Stream<Place> specPlaces;
    @ public invariant specPlaces.count() <= size &&
    @    specPlaces().matchAll(p -> p.getShip() == this) &&
    @    (specPlaces.count() == size  ==> Place.isSeq(specPlaces)); @*/

  private final List<Place> places;
  //@ private represents specPlaces <- places.stream().sorted(cmp);
  /*@ private final model Comparator<Place> cmp = new Comparator<>() {
    @     public int compare(Place p1, Place p2) {
    @         int diff = p1.getX() – p2.getX();
    @         return diff != 0 ? diff : p1.getY() – p2.getY();
    @     }
    @ }; @*/

  /*@ requires 2 <= size && size <= 5;
    @ assignable this.size, specPlaces;
    @ ensures this.size == size;
    @ ensures specPlaces.count() == 0; @*/
  public Ship(int size) { ... }

  //@ ensures \result == size;
  public /*@ pure @*/ int size() { ... }

  /*@ requires specPlaces.count() > 0;
    @ ensures \result == specPlaces.findFirst(); @*//
  public /*@ pure @*/ Place head() { ... }

  /*@ requires specPlaces.count() > 0;
    @ ensures \result == specPlaces.skip(specPlaces.count()-1).findFirst(); */
  public /*@ pure @*/ Place tail() { ... }

  /*@ requires specPlaces.count() > 0;
    @ ensures \result == specPlaces.mapToInt(p ->
    @   p.getY()).distinct().count() == 1; @*/
  public /*@ pure @*/ boolean isHorizontal() { ... }

  /*@ requires specPlaces.count() > 0;
    @ ensures \result == specPlaces.mapToInt(p ->
    @   p.getX()).distinct().count() == 1; @*/
  public /*@ pure @*/ boolean isVertical() { ... }

  /*@ ensures StreamSupport.stream(\result.spliterator(), false)
    @    equals(specPlaces); @*/
  public /*@ pure @*/ Iterable<Place> places() { ... }

  //@ ensures \result == specPlaces.allMatch(p -> p.isHit());
  public /*@ pure @*/ boolean isSunk() { ... }

  /*@ requires place.ship() == this;
    @ requires specPlaces.noneMatch(p -> p.equals(place));
    @ requires specPlaces.count() < size;
    @ requires specPlaces.count() == size – 1 ==>
    @   Place.isSeq(specPlaces.concat(Stream.of(place))); @*/
    @ assignable specPlaces;
    @ ensures toSet(specPlaces).equals(\old(toSet(
    @    specPlaces.concat(Stream.of(place))))); @*/
  public void addPlace(Place place) { ... }
}
```