Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2

Patrice Chalin[1], Joseph R. Kiniry[2], Gary T. Leavens[3], and Erik Poll[4]

[1] Concordia University, Montréal, Québec, Canada
[2] University College Dublin, Ireland
[3] Iowa State University, Ames, Iowa, USA
[4] Radboud University Nijmegen, the Netherlands

Abstract. Many state-based specification languages, including the Java Modeling Language (JML), contain at their core specification constructs familiar to most undergraduates: e.g., assertions, pre- and postconditions, and invariants. Unfortunately, these constructs are not sufficiently expressive to permit formal modular verification of programs written in modern object-oriented languages like Java. The necessary extra constructs for specifying an object-oriented module include (perhaps the less familiar) frame properties, datagroups, and ghost and model fields. These constructs help specifiers deal with potential problems related to, for example, unexpected side effects, aliasing, class invariants, inheritance, and lack of information hiding. This tutorial paper focuses on JML's realization of these constructs, explaining their meaning while illustrating how they can be used to address the stated problems.

1 Introduction

Textbooks on program verification typically explain the notions of pre- and postconditions, loop invariants, and so on for toy programming languages. The goal of this paper is to explain some of the more advanced concepts that are necessary in order to allow the formal modular verification of (sequential) programs written in a popular mainstream object-oriented language: Java. The Java Modeling Language (JML) [BCC+05,LBR06,LPC+06], a Behavioral Interface Specification Language (BISL) [Win90] for Java, will be our notation of choice for expressing specifications.

The reader is assumed to be familiar with the basics of Design by Contract (DBC) [Mey97] or Behavioral Interface Specifications (BISs) and the central role played by assertions in these approaches. Readers without this background may wish to consult one of several books or articles offering tutorials on the subject [Hoa69,LG01,MM02,Mey92,Mey97,Mor94]. A tutorial that explains these basic ideas using JML is also available [LC05].

1.1 Approaches to verification

Tools useful for checking that JML annotated Java modules meet their specifications fall into two main categories:[5]

- runtime assertion checking (RAC) tools, and
- static verification (SV) tools.

These categories also represent two complementary forms of assertion checking, the foundations of which were laid out before the 1950s in the pioneering work of Goldstine, von Neumann and Turing [Jon03]. Runtime assertion checking involves the testing of specifications during program execution; any violations result in special errors being reported. The idea of checking *contracts* at runtime was popularized by Eiffel [Mey97] as of the late 80s; other early work includes Rosenblum's APP annotation language for C [Ros92,Ros95]. The main RAC tool for JML is `jmlc` [CL02]. RAC support for JML is also planned for the next release of the Jass tool [BFMW01].

In static verification, logical techniques are used to prove, before runtime, that no violations of specifications will take place at runtime. The adjective *static* emphasizes that verification happens by means of a static analysis of the code, i.e., without running it. Program verification tools supporting JML include JACK [BRL03], KeY [ABB+05], Krakatoa [MPMU04], LOOP [BJ01], and Jive [MPH00]. In this paper we will focus on ESC/Java2 [CK04], the main (extended) static checker for JML.

RAC and SV tools have complementary strengths. Compared to runtime assertion checking, static verification often provides stronger guarantees and it can give them earlier. However, these advantages come at a price: SV tools generally require fairly complete specifications not only for the module being checked, but also for the modules and libraries that it depends on. Furthermore, in order to be effective and keep false positives to a minimum, SV tools require specifications to make use of some of the advanced features described in this paper.

1.2 Outline

The remainder of the paper is organized as follows. The basic notation used in JML for method contracts and invariants is covered in Section 2. Section 3 explains frame properties, and Section 4 model fields. The treatment of behavioral subtyping is given in Section 5. Section 6 explains ghost fields. Section 7 introduces the JML notations that deal with ownership and aliasing. Finally, conclusions and related work are given in Section 8.

2 JML Basics: Pre- and Postconditions, and Invariants

This section examines the specification and implementation of various kinds of clocks. In doing so, we review basic concepts such as method contracts and class invariants and introduce their JML notation.

[5] There are also several other kinds of tool available for use with JML [BCC+05].

2.1 Method contracts

We begin with the specification of a `TickTockClock` as given in Fig. 1. This specification illustrates basic method contracts formed by:

- preconditions (introduced by the `requires` keyword), and
- postconditions (`ensures`).

An example of such a contract is found in the specification of the method `getSecond()` on lines 23–24. The JML specification for each method is written in front of the method itself, and is found in stylized Java comments that begin with an at-sign (‘`@`’).

A method contract without an explicit `requires` clause has an implicit precondition of *true*. Thus, such a method imposes no requirements on its callers. This default means that the `requires` clause written for `getHour()` could have been omitted entirely. Similarly, the default postcondition when none is explicitly given in an `ensures` clause is also *true*, which says that the method makes no guarantees to its caller. The constructor (on lines 12–15) and the method `getMinute()` (on lines 21–22) are examples of class members with implicit `requires` clauses.

Note that assertion expressions appearing in `requires` and `ensures` clauses are written using a Java-like syntax. In postconditions of (non-`void`) methods, `\result` can be used to refer to the value being returned by the method. The only other JML specific operator used in this clock specification is the `\old`() operator, which is used in an `ensures` clause of `tick` (on lines 29–31). The expression `\old`(e) refers to the value of e in the method's pre-state, i.e., the state just before the method is executed.

Preconditions and postconditions are often split over multiple `requires` and `ensures` clauses, as illustrated for the postcondition of `getSecond()` (on lines 23–24). Multiple `ensures` clauses, or multiple `requires` clauses, are equivalent to a single clause consisting of the conjunction (`&&`) of their respective assertions.

Method contracts, like the contract of `tick()` on lines 27–37 of Fig. 1, are written as one or more *specification cases* combined with the keyword `also`. Each specification case is a “mini-contract” in itself, having a precondition and postcondition (either explicit or implicit) as well as other clauses that are covered below. Use of specification cases allows developers to structure their specifications and to (literally) break it up into (generally) distinct cases.

The contract for `tick()`, which is somewhat contrived for illustrative purposes, highlights to clients that its behavior essentially has two cases of interest. Either

- seconds are less than 59 and the seconds are incremented by one, or
- seconds are at 59 and they will be wrapped back to 0.

We note in passing that the specification of `tick()` is incomplete, as it might be during the development of the `TickTockClock` class. Informal comments, like the one on line 36, are useful for remembering what remains to be formalized or to avoid formalization (e.g., if it is too costly), although they do not help in verification.

```
1  public class TickTockClock {
2    //@ public model JMLDataGroup _time_state;
3
4    //@ protected invariant 0 <= hour && hour <= 23;
5    protected int hour; //@ in _time_state;
6    //@ protected invariant 0 <= minute && minute <= 59;
7    protected int minute; //@ in _time_state;
8    //@ protected invariant 0 <= second && second <= 59;
9    protected int second; //@ in _time_state;
10
11   //@ ensures getHour() == 12 && getMinute() == 0 && getSecond() == 0;
12   public /*@ pure @*/ TickTockClock() {
13       hour = 12; minute = 0; second = 0;
14   }
15
16   //@ requires true;
17   //@ ensures 0 <= \result && \result <= 23;
18   public /*@ pure @*/ int getHour() { return hour; }
19
20   //@ ensures 0 <= \result && \result <= 59;
21   public /*@ pure @*/ int getMinute() { return minute; }
22
23   //@ ensures 0 <= \result;
24   //@ ensures \result <= 59;
25   public /*@ pure @*/ int getSecond() { return second; }
26
27   /*@  requires   getSecond() < 59;
28     @  assignable hour, minute, second; // NB for expository purposes only
29     @  assignable _time_state;
30     @  ensures    getSecond() == \old(getSecond() + 1) &&
31     @             getMinute() == \old(getMinute()) &&
32     @             getHour() == \old(getHour());
33     @ also
34     @  requires   getSecond() == 59;
35     @  assignable _time_state;
36     @  ensures    getSecond() == 0;
37     @  ensures    (* hours and minutes are updated appropriately *);
38     @*/
39   public void tick() {
40       second++;
41       if (second == 60) { second = 0; minute++; }
42       if (minute == 60) { minute = 0; hour++; }
43       if (hour == 24)   { hour = 0; }
44   }
45 }
```

Fig. 1. JML specification for `TickTockClock`. The datagroup `_time_state`, the associated `assignable` clauses and `in` clauses are explained later, in Section 3.

2.2 Purity

In the DBC approach, only query methods can be used in assertion expressions because they are required to be side-effect free [Mey97]. The corresponding concept in JML is known as method *purity*; pure methods are not allowed to have side effects, and pure constructors can only assign to the fields of the object they are initializing. Purity is statically checked by the JML tools. The restriction that only methods declared as `pure` can be used in assertion expressions is also checked statically. E.g., since the method `getSecond()` is declared `pure`, it is legal to make use of it in the postcondition of `tick()`.

Notice that the `TickTockClock` constructor is declared as `pure` despite the fact that it assigns to the fields `hour`, `minute` and `second`. Such instance field assignments are permitted inside the bodies of constructors because they are benevolent side-effects—i.e., that have no observable effect on clients. On the other hand, a pure constructor would not be permitted to assign to a static field. Purity, and particularly variants in the strength (restrictiveness) of its definition are a subject of active research—e.g., a stronger notion of purity than that of JML has been proposed by Darvas and Müller [DM05]. On the other hand, purity is often too strong [LCC+05], and so a notion of "observational purity" that permits benevolent side effects (such as updates to caches) is also under consideration [BSS04,Nau05].

2.3 Lightweight vs. heavyweight

JML actually has two kinds of specification cases: lightweight and heavyweight. Lightweight specification cases are useful when giving partial specifications, and in practice are often used with ESC/Java2. To convey that one is intending to give a complete specification for some precondition, one would use a heavyweight specification case. Such heavyweight specification cases are often used with runtime assertion checking.

The specification cases of the `tick()` method are lightweight. An example use of heavyweight specification cases is found on the `setTime()` method of the `SettableClock` class given in Fig. 2. A heavyweight specification case is easily recognized by the use of a "`behavior`" keyword at the beginning of the case. The contract of `setTime()` illustrates the two kinds of heavyweight specification cases most often used. The first specification case uses the `normal_behavior` keyword and it describes the intended behavior of the method when it returns normally. The second specification case uses the `exceptional_behavior` keyword and it describes the intended behavior of the method when the method raises an exception. The latter case is described at greater length in Section 2.4. Notice that the heavyweight specification cases of `setTime()` start with `public`. This means that the specification cases are visible to clients, and hence, for example, will be included as a part of client visible documentation generated using JmlDoc [BCC+05]. It also means that these specification cases cannot refer to `private` or `protected` fields.

```
1  class SettableClock extends TickTockClock {
2
3      // ...
4
5      /*@ public normal_behavior
6        @   requires   0 <= hour && hour <= 23 &&
7        @              0 <= minute && minute <= 59;
8        @   assignable _time_state;
9        @   ensures    getHour() == hour &&
10       @              getMinute() == minute && getSecond() == 0;
11       @ also
12       @  public exceptional_behavior
13       @   requires     !(0 <= hour && hour <= 23 &&
14       @                  0 <= minute && minute <= 59);
15       @   assignable   \nothing;
16       @   signals      (IllegalArgumentException e) true;
17       @   signals_only IllegalArgumentException;
18       @*/
19     public void setTime(int hour, int minute) {
20       if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
21         throw new IllegalArgumentException();
22       }
23       this.hour = hour;
24       this.minute = minute;
25       this.second = 0;
26     }
27 }
```

Fig. 2. JML specification for `SettableClock`

Contracts built from lightweight specification cases have fewer keywords and mandatory clauses. In particular, the visibility of a lightweight specification case cannot be given explicitly since, by definition, its visibility is the same as the visibility of the method it is attached to. The method contracts in `TickTockClock` are all examples of lightweight method specifications.

2.4 Exceptions and exceptional postconditions

JML distinguishes two kinds of postcondition:

- normal postconditions, expressed by means of `ensures` clauses, that must hold when a method terminates normally, and
- exceptional postconditions, expressed by means of `signals` clauses, that must hold when a method terminates with an exception.

The exceptional specification case of `SettableClock.setTime()` is interpreted as follows: if `hour` and `minute` are not within their valid ranges, then the method will raise an `IllegalArgumentException` and the system state will be left unchanged.

Notice that in the `TickTockClock` class there are no Java `throws` clauses. Still, Java permits the constructor and any of the methods of this class to throw a `RuntimeException`—one commonly raised runtime exception is `NullPointerException`. JML is more strict when it comes to declaring runtime exceptions: whereas Java allows any constructor or method to throw a runtime exception, JML only allows this if the exception is listed in the method's throws clause, or in the method contract's `signals_only` clause. `SettableClock.setTime()` illustrates use of the latter. Therefore, constructors or methods without an explicit throws clause have an implicit exceptional postcondition of `signals (Exception) false`. So the specification in Fig. 1 rules out the generation of any runtime exceptions, making the specification much stronger than it might appear at first sight. However, JML, like Java, makes a distinction between exceptions and errors; since Java's type `Error` is not a subtype of `Exception`, JML specifications do not say anything about virtual machine errors, such as running out of memory [PH97].

2.5 Instance and static invariants (and the callback problem)

A JML `invariant` clause declared with a `static` modifier is called a *static invariant*. Static invariants express properties which must hold of the static attributes of a class. An assertion that appears in a non-static `invariant` clause is called a *instance invariant* or an *object invariant*. Note that while this terminology is contrary to the literature, it is more accurate with respect to the nomenclature of Java. In this paper, an unqualified use of the term "invariant" will refer to an "object invariant."

The semantics of object invariants is more involved than most specifiers expect, especially for newcomers to the field of object-oriented specification. Hence, while this issue has been widely known for quite some time [Szy98], we believe it is worth a brief explanation. Intuitively, an object invariant:

- has to be established by constructors—i.e., it is implicitly included in the postcondition of constructors;
- can be assumed to hold on entry to methods, but methods must also reestablish it on exit. Hence, the invariant is implicitly included in the preconditions, and (normal and exceptional) postconditions of methods.

This intuition may suggest that the notion of object invariant is not really necessary, but rather that it just provides a convenient shorthand. This idea is a common misconception, as there is more to the notion of invariant than the intuitions summarized above. One difference is that invariants apply to all subtypes through specification inheritance (Section 5), whereas predicates that just happen to appear in all pre-and post conditions are not inherited as part of the specification of any new methods that may be added in a subtype.

One other issue is related to callbacks. For example, suppose that the `tick` method called another method at a program point where its invariant is broken, such as the call to `canvas.paint()` in the following:

```
public void tick() {
  second++;
  // object invariant might no longer hold
  canvas.paint();
  /* ... */
}
```

It would then be reasonable for the canvas to invoke, e.g., the `getSecond()` method of the current clock object, performing a so-called callback. However, since the invariant of this clock object is broken, its behavior is unconstrained, in particular because the preconditions of all methods (which implicitly include the object invariant) are all false.

To avoid such problems, the invariant not only has to be re-established at the end of each method, but also at those program points where a (non-helper) method is invoked. These program points—i.e., all program points at which a method invocation starts or ends—are called the *visible states*. The visible state semantics for invariants says that all invariants of all objects must hold at these visible states. This semantics is very strong and in many cases overly restrictive. Less restrictive, but still sound, approaches are still a hot topic of ongoing research. A more thorough discussion of this problem and a proposed solution for JML is given in [MPHL05]; alternative solutions are explored elsewhere [BDF+04,HK00,JLPS05,MHKL05].

3 Frame properties

In traditional specifications that give pre- and postcondition for methods (or procedures) one often uses the convention that any variables not mentioned in the postcondition have not been changed. This approach is *not* workable for realistic object-oriented programs. For example, consider the method `tick()` in Fig. 1. This method may modify the three private fields `second`, `minute` and `hour`, but these do not appear in the postcondition. Rewriting the specification so it does mention these fields is clearly not what we would want, since in the specification of this public method we do not want to refer to private fields.

A JML `assignable` clause is used in a method contract to specify which parts of the system state *may* change as the result of the method execution. This is the so-called *frame property* [BMR95]. Any location outside the frame property is guaranteed to have the same value after the method has executed (called the post-state) as it did before the method executed (in the pre-state). The notion of *datagroup* [Lei98] allows us to abstract away from private implementation details in frame properties and provides flexibility in specifications. This section explains these notions and the need for them.

An `assignable` clause specifies that a method may change certain fields without having to specify how they might change. So the specification of the method `tick()` could include

```
assignable hour, minute, second;
```

to state that it may modify these three fields, without having to mention the fields in the postcondition. If no `assignable` clause is given for a non-pure method, then it has the default frame condition `assignable \everything`. However, pure methods (Section 2.2) have a default frame of `assignable \nothing`.

Object-oriented languages such as Java require some means for abstraction in assignable clauses. E.g., the first `assignable` clause for `tick()` given above leaves a lot to be desired. Firstly, it exposes implementation details, because it mentions the names of protected fields. Secondly, the specification is overly restrictive for any future subclasses. By the principle of *behavioral subtyping*, discussed in more detail in Section 5, the implementation of `tick()` in any future subclass of `TickTockClock` has to meet the specification given in `TickTockClock`. This means that the method body can only assign to the three fields of `TickTockClock`, which is far too restrictive in practice. To give a concrete example, suppose we introduce a subclass `TickTockClockWithDate` of `TickTockClock` that, in addition to keeping the time, also keeps track of the current date. Clearly such a subclass will introduce additional fields to record the date and `tick` will have to modify these fields when the end of a day is reached; however, the assignable clause given above will not allow these fields to be changed, as they are not explicitly listed.

Datagroups [Lei98] provide a solution to this problem. The idea is that a datagroup is an abstract piece of an object's state that may still be extended by future subclasses. The specification in Fig. 1 declares a (public) datagroup `_time_state` and declares that the three (private) fields belong to this datagroup. This datagroup is (partially) used to specify `tick()`. This avoids exposing any private implementation details, and subclasses of `TickTockClock` may extend the datagroup with additional fields it introduces.

Datagroups may be nested by using the `in` clause to say that one datagroup is part of another one. The JML specification for `java.lang.Object` declares a datagroup named `objectState`. Since this datagroup is inherited by all other classes, as a convention one can use `objectState` in any class to describe what constitutes the 'state' of an object of that class. Had we followed this convention then, e.g., we would have declared the `_time_state` datagroup to be in `objectState`.

Finally we note that, although `assignable` clauses are needed when doing program verification, they are *not* currently used during runtime assertion checking. (The RAC tool checks `assignable` clauses statically and does not check them at runtime.)

4 Model fields

Model fields [CLSE05] are closely related to the notion of data abstraction proposed by Hoare [Hoa72]. A model field is a specification-only field that provides an abstraction of (part of) the concrete state of an object. The specification in Fig. 3 illustrates the use of a model field `_time` that represents the number of seconds past midnight. Notice how this abstraction allows for a brief but complete specification of the method `tick()`. The `represents` clause of line 3 relates the model field to its concrete representation, in this case as a function of `hour`, `minute` and `second`. Hence, the `represents` clause defines the representation function of `_time`. (In its most general form, JML also permits `represents` clauses that are relational [LPC+06], but we do not discuss these here.)

Note that the `_time` model field is public, and hence visible to clients, though its representation is not. The `represents` clause must be declared private, because it refers to private fields. For every model field there is an associated datagroup, so that the model field can also be used in `assignable` clauses. In fact, a field of type `JMLDataGroup` is a degenerate model field that holds no information.

A difference between model fields for objects and the traditional notion of abstract value for abstract data types is that an object can have several model fields, providing abstractions of different aspects of the object. For instance, the specification of `AlarmClock` (a subclass of `Clock`, given in Fig. 4) uses two model fields, one for the current time, which it inherits from `Clock`, and one for the alarm time.

Model fields are especially useful in the specification of Java interfaces, as interfaces do not contain any concrete representation we can refer to in specifications. We can declare model fields in a Java interface then every class that implements the interface can define its own `represents` clause relating this abstract field to its concrete representation. For a more extensive discussion of model fields see [CLSE05]. Cok discusses how model fields are treated in ESC/Java2 [Cok05], while Leino and Müller have recently worked on handling model fields in the context of verification [LM06].

5 Behavioral subtyping and specification inheritance

JML enforces *behavioral subtyping* [Ame90,DL96,LD00,LW95,LW94,Mey97]: instances of a given type T must meet the specifications of each of type T's supertypes. This ensures Liskov's "substitution principle" [Lis88], i.e., it ensures that using an object of a subclass in a place where an object of the superclass is expected does not cause any surprises, ensuring that the introduction of new subclasses does not break any existing code. This idea is also known as supertype abstraction [Lea90,LW95].

```
1  public class Clock {
2    //@ public model long _time;
3    //@ private represents _time = second + minute*60 + hour*60*60;
4
5    //@ public invariant _time == getSecond() + getMinute()*60 + getHour()*60*60;
6    //@ public invariant 0 <= _time && _time < 24*60*60;
7
8    //@ private invariant 0 <= hour && hour <= 23;
9    private int hour; //@ in _time;
10   //@ private invariant 0 <= minute && minute <= 59;
11   private int minute; //@ in _time;
12   //@ private invariant 0 <= second && second <= 59;
13   private int second; //@ in _time;
14
15   //@ ensures _time == 12*60*60;
16   public /*@ pure @*/ Clock() { hour = 12; minute = 0; second = 0; }
17
18   //@ ensures 0 <= \result && \result <= 23;
19   public /*@ pure @*/ int getHour() { return hour; }
20
21   //@ ensures 0 <= \result && \result <= 59;
22   public /*@ pure @*/ int getMinute() { return minute; }
23
24   //@ ensures 0 <= \result && \result <= 59;
25   public /*@ pure @*/ int getSecond() { return second; }
26
27   /*@ requires   0 <= hour && hour <= 23;
28     @ requires   0 <= minute && minute <= 59;
29     @ assignable _time;
30     @ ensures    _time == hour*60*60 + minute*60;
31     @*/
32   public void setTime(int hour, int minute) {
33     this.hour = hour; this.minute = minute; this.second = 0;
34   }
35
36   //@ assignable _time;
37   //@ ensures _time == \old(_time + 1) % 24*60*60;
38   public void tick() {
39     second++;
40     if (second == 60) { second = 0; minute++; }
41     if (minute == 60) { minute = 0; hour++; }
42     if (hour == 24)   { hour = 0; }
43   }
44 }
```

Fig. 3. Example JML specification illustrating the use of model fields.

For example, consider the class `AlarmClock` in Fig. 4. Because `AlarmClock` is a subtype of `Clock`, it inherits all the specifications of `Clock`, i.e., all invariants specified for `Clock` also apply to `AlarmClock`, and any (overriding) method in `AlarmClock` has to meet the specification for the corresponding method in `Clock`. For example, the overriding `AlarmClock` method `tick()` has to meet the specification given for it in `Clock`. Note that any methods which are not overridden have to be re-verified, to ensure that they maintain any additional invariants of the subclass. ([RL00] investigates ways to avoid some of this re-verification.)

When it comes to method specifications, behavioral subtyping requires that the specification of an overriding method m must refine that of its supertypes in the sense that whenever a supertype's precondition for m is satisfied, then the supertype's postcondition for m must hold. It follows that the preconditions of an overriding method may only be weaker. Furthermore, whenever an overridden method's precondition is satisfied then the postcondition of the overriding method must imply the postcondition of the overridden method. One way to achieve this is would be to allow a subtype to give a new specification for a method—effectively overriding the one in the supertype—and then prove the necessary refinement relationship. Instead, JML uses the principle of *specification inheritance* for method specifications [DL96]: all specification cases written for an overriding method are "conjoined" (using `also`) with the specification cases of the method(s) being overridden. Specification inheritance guarantees that the overriding method obeys all the inherited specification cases and thus that the method satisfies a refinement of the inherited specifications. This automatically makes all subtypes behavioral subtypes and thus validates the principle of supertype abstraction.

The meaning of specification cases conjoined by `also` can be a bit subtle. However, it is easiest to just keep in mind that all specification cases of all inherited methods have to each be obeyed by a method. If, for a given method, the subtype and supertypes all specify the same precondition and assignable clause, then the conjoined specification will be equivalent to a single specification case whose precondition and assignable clause are the same as in the individual specification cases, and with a postcondition that is the conjunction of the postconditions in the individual specification cases. If different preconditions are given in a sub- and supertype the meaning of the conjoined specification cases is more involved: the precondition of the conjoined specification will effectively be the disjunction of the preconditions from the individual specification cases, and the postcondition of the conjoined specification will effectively be a conjunction of implications, where each precondition (wrapped in `\old()`) implies the corresponding postcondition. This effective postcondition is slightly weaker than the conjunction of the postconditions, since each postcondition only has to apply in case the corresponding precondition was satisfied [DL96].

Before closing this section we point out that the `alarm` field (line 13) and `alarm` parameter (line 15) of the `AlarmClock` class are explicitly declared to be non-null instances of `AlarmInterface`. While this is unnecessary (since declarations of reference types are non-null by default in JML [LCC+05,Cha06]), it is

```
class AlarmClock extends Clock {
  //@ public model int _alarmTime;
  //@ private represents _alarmTime = alarmMinute*60 + alarmHour*60*60;

  //@ public ghost boolean _alarmOn = false; //@ in _time;

  //@ private invariant 0 <= alarmHour && alarmHour <= 23;
  private int alarmHour; //@ in _alarmTime;

  //@ private invariant 0 <= alarmMinute && alarmMinute <= 59;
  private int alarmMinute; //@ in _alarmTime;

  private /*@ non_null @*/ AlarmInterface alarm;

  public /*@ pure @*/ AlarmClock(/*@ non_null @*/ AlarmInterface alarm) {
    this.alarm = alarm;
  }

  /*@ requires   0 <= hour && hour <= 23;
    @ requires   0 <= minute && minute <= 59;
    @ assignable _alarmTime;
    @*/
  public void setAlarmTime(int hour, int minute) {
    alarmHour = hour;
    alarmMinute = minute;
  }

  // spec inherited from superclass Clock
  public void tick() {
    super.tick();
    if (getHour() == alarmHour & getMinute() == alarmMinute & getSecond() == 0) {
      alarm.on();
      //@ set _alarmOn = true;
    }
    if ((getHour() == alarmHour & getMinute() == alarmMinute+1 & getSecond() == 0) ||
        (getHour() == alarmHour+1 & alarmMinute == 59 & getSecond() == 0) ) {
      alarm.off();
      //@ set _alarmOn = false;
    }
  }
}
```

Fig. 4. Example JML specification illustrating the concepts of specification inheritance and ghost fields.

```
1  public interface AlarmInterface {
2    public void on();
3    public void off();
4  }
```

Fig. 5. Interface of the alarm used in `AlarmClock`

also harmless and can in fact be helpful to JML newcomers. Though we will not have the need in our examples, declarations that may be null must be annotated with the `nullable` modifier.

6 Ghost fields

Like model fields, ghost fields are specification-only fields, so they cannot be referred to by Java code. While a model field provides an abstraction of the existing state, a ghost field can provide some additional state, which may—or may not—be related to the existing state. Unlike a model field, a ghost field can be assigned a value. This is done by a special `set` statement that must be given in a JML annotation. Before we discuss the difference between model and ghost fields in more detail, let us first look at an example of the use of a ghost field.

Suppose that we want to convince ourselves that the implementation of `AlarmClock` will not invoke the method `alarm.on()` twice in a row, or the method `alarm.off()` twice in a row, but that it will always call `alarm.on()` and `alarm.off()` alternately. (One could add JML contracts to `AlarmInterface` to specify this requirement, but we will not consider that here.)

The state of an `AlarmClock` object does not record if the associated `alarm` is ringing or not, nor does it record which method it has last invoked on `alarm`. For the purpose of understanding the behavior of the `AlarmClock`, and possibly capturing this understanding in additional JML annotations, it may be useful to add an extra boolean field to the state that records if the associated alarm is ringing. In Fig. 4, we have declared a boolean ghost field `_alarmRinging`. Two assignments to this field are included in the method `tick()`. The assignments ensure that the field is true when the alarm ringing and false otherwise. A subtle issue here is that `_alarmRinging` has to be included in the datagroup associated with `_time`. This is because—by the principle of specification inheritance—the method `tick()` is only allowed to have side effects on `_time`. Since `tick()` assigns to `_alarmRinging`, the field has to be included in this datagroup. (As was mentioned in Section 3, we could have instead declared `_time` to be in `objectState`, and used `objectState` in the assignable clause of `tick()`. It then would have been more natural to declare `_alarmRinging` to be in `objectState`.)

One can now try to capture the informal requirement that "the alarm will ring for the minute that follows the specified alarm time," by formulating invariants relating the new ghost field `_alarmRinging` to the 'real' state of the `AlarmClock`. There are many ways to express such a relation, for instance using the following as the invariant:

```
_alarmRinging <==> _alarmTime <= _time && _time < alarmTime + 60;
```

Verification by ESC/Java2 will immediately point out that these invariants may be violated, namely by invocations of `setTime` and `setAlarmTime`. This highlights a potential weakness in the implementation: relying on the comparison of the current time and the alarm time in the decision to turn the alarm off might result in unwanted behavior. The alarm could be turned on twice in a row, or turned off twice in a row. Also, the alarm could ring for longer than 60 seconds, if one of these times is changed while the alarm is ringing.

An improvement in the implementation is to count down the number of seconds left until the alarm is disabled and use this count as a basis for switching off the alarm, rather than relying on a comparison of the current time and the alarm time.

```
/** The number of seconds remaining to keep ringing the alarm.
  * If zero, the alarm is silent (off). */
//@ private invariant 0 <= alarmSecondsRemaining &&
//@                   alarmSecondsRemaining <= 60;

/*@ private invariant _alarmRinging
  @                   <==> alarmSecondsRemaining > 0; @*/
private int alarmSecondsRemaining = 0; //@ in _time;
...

public boolean tick() {
  super.tick();
  if (alarmSecondsRemaining > 0) {
    alarmSecondsRemaining--;
    if (alarmSecondsRemaining == 0) {
      alarm.off();
      //@ set _alarmRinging = false;
    }
  } else if (getHour() == alarmHour &
             getMinute() == alarmMinute) {
           alarm.on();
           alarmSecondsRemaining = 60 - getSecond();
           //@ set _alarmRinging = true;
  }
}
```

Now that we have a close correspondence between the ghost field `_alarmRinging` and the field `alarmSecondsRemaining`, one could choose to replace the ghost field by a model field:

```
/*@ public model boolean _alarmRinging; in _time;
  @ private represents _alarmRinging
  @                       <- alarmSecondsRemaining > 0;
  @*/
```

Of course, one could also choose to turn the ghost field into a real field. This would make the implementation simpler to understand.

Ghost vs. model fields. To recap, the crucial difference between a ghost and a model field is that a ghost field extends the state of an object, whereas a model field is an abstraction of the existing state of an object. A ghost field may be assigned to in annotations using the special `set` keywords. A model field cannot be assigned to, but changes value automatically whenever some of the state that it depends on changes, as laid down by the representation relation.

Since ghost fields are only changed by `set` statements, they are only changed under program control. Model fields, however, potentially change their values whenever the concrete fields they depend on change. As Leino and Müller recently noted [LM06], such instantaneous changes to model fields are not necessarily sensible, because the computation of the model fields may assume that various invariants hold.

7 Aliasing

The potential of aliasing is a major complication in program verification, and indeed a major source of bugs in programs. To illustrate the issue, Fig. 6 shows `DigitalDisplayClock`, which uses an integer array `time` of length 6 to represent time (line 13). For the correct functioning of the clock it will be important that this array is not aliased by a field outside of the class. If the array is aliased, code outside of this class could alter `time` and break the invariants for the array [NVP98]. Indeed, the fact that the (private) invariants depends on the array `time` already suggest that the field needs to be alias-protected.

By inspecting the entire code of the class, it is easy to convince oneself that references to this array are not leaked. However, this does not guarantee that a subclass does not introduce ways to leak references to `time`. For example, the subclass `BrokenDigitalDisplayClock` in Fig. 7 breaks the guarantee that `time` will not be aliased.

There has been considerable work on extending programming languages with some form of ownership (also known as confinement). JML includes support for the universe type system [MPHL03] as a way to specify and enforce ownership constraints. As is illustrated in Fig. 6 line 13, the `time` array is declared as a `rep`-field[6] hence forbidding `time` from being aliased outside the object. The typechecker incorporated in the JML compiler will, e.g., warn that the class `BrokenDigitalDisplayClock` in Fig. 7 is not well-typed because it breaks the guarantee that `time` will not be aliased outside this class. Verification with ESC/Java2 does not yet take universes into account and this is still the subject of ongoing work.

[6] `rep` is short for representation.

```
1  public class DigitalDisplayClock {
2    //@ public model long _time;
3    //@ private represents _time = getSecond()+getMinute()*60+getHour()*60*60;
4
5    //@ protected invariant time.length == 6;
6    //@ protected invariant 0 <= time[0] && time[0] <= 9; // sec
7    //@ protected invariant 0 <= time[1] && time[1] <= 5; // sec
8    //@ protected invariant 0 <= time[2] && time[2] <= 9; // min
9    //@ protected invariant 0 <= time[3] && time[3] <= 5; // min
10   //@ protected invariant 0 <= time[4] && time[4] <= 9; // hr
11   //@ protected invariant 0 <= time[5] && time[5] <= 2; // hr
12   //@ protected invariant time[5] == 2 ==> time[4] <= 3; // hr
13   protected /*@ non_null rep @*/ int[] time; // NB rep modifier
14   /*@ pure @*/ public DigitalDisplayClock() {
15     { time = new rep int [6]; }   // NB rep modifier
16
17   //@ ensures 0 <= \result && \result <= 23;
18   public /*@ pure @*/ int getHour() { return  time[5]*10 + time[4]; }
19
20   //@ ensures 0 <= \result && \result <= 59;
21   public /*@ pure @*/ int getMinute() { return time[3]*10 + time[2]; }
22
23   //@ ensures 0 <= \result && \result <= 59;
24   public /*@ pure @*/ int getSecond() { return time[1]*10 + time[0]; }
25
26   /*@ requires   0 <= hour && hour <= 23 && 0 <= minute && minute <= 59;
27     @ assignable _time;
28     @ ensures    getHour()==hour && getMinute()==minute && getSecond()==0;
29     @*/
30   public void setTime(int hour, int minute) {
31       time[5] = hour / 10;    time[4] = hour % 10;
32       time[3] = minute % 10;  time[2] = minute % 10;
33       time[1] = 0 ;           time[0] = 0;
34   }
35
36   //@ assignable _time;
37   //@ ensures _time == (\old(_time)+1) % 24*60*60;
38   public void tick() {
39       time[0]++;
40       if (time[0] == 10) { time[0] = 0; time[1]++; }
41       if (time[1] == 6)  { time[1] = 0; time[2]++; } // minute passed
42       if (time[2] == 10) { time[2] = 0; time[3]++; }
43       if (time[3] == 6)  { time[3] = 0; time[4]++; } // hour passed
44       if (time[4] == 10) { time[4] = 0; time[3]++; }
45       if (time[5] == 2 & time[4] == 4)
46           { time[5] = 0; time[4] = 0; } // day passed
47   }
48 }
```

Fig. 6. Clock implementation using an array and the universe type system to ensure that references to this array are not leaked outside the current object.

```
1  class BrokenDigitalDisplayClock extends DigitalDisplayClock {
2    //@ requires time.length == 6;
3
4    public BrokenDigitalDisplayClock( /*@ non_null @*/ int[] time) {
5      this.time = time; // illegal!
6    }
7
8    public /*@ pure @*/ int[] expose() { return time; } // illegal!
9  }
```

Fig. 7. A subclass of `DigitalDisplayClock` which breaks encapsulation of the private array `time`, both by its constructor, which imports a potentially aliased reference, and the method `expose`, which exports a reference to `time`.

8 Conclusions

Preconditions, postconditions and invariants alone are insufficient to accurately specify object-oriented programs. This paper has illustrated some of the more advanced specification constructs of the JML specification language, notably: frame conditions, datagroups, model and ghost fields, and support for alias control.

A language extension to C# that is similar in purpose and scope to JML is the Spec# specification language [BLS04]. Like JML, Spec# enjoys tool support for runtime checking and static verification, the latter being provided by the Boogie program verifier. Spec# and JML share similar basic and advanced language constructs, although details vary. In particular, Spec# provides a novel methodology to cope with object invariants [BDF+04].

As a final note, we point out that the question of which constructs are necessary and sufficient for the specification of mainstream object-oriented programs is far from settled. Even the semantics for some of the basic, let alone advanced, features discussed in this paper are still the subject of active research as is clear from the references given to very recent work.

Acknowledgments Thanks to David Cok of Eastman Kodak Company for his comments and feedback on this paper. The work of Joseph Kiniry and Erik Poll is funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The work of Gary Leavens was funded by the US National Science Foundation under grants CCF-0428078 and CCF-0429567. Patrice Chalin was funded in part by the Natural Sciences and Engineering Research Council of Canada under grant 261573-03.

References

[ABB+05] W. Ahrendt, Th. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool.

Software and System Modeling, 4:32–54, 2005.

[Ame90] P. America. Designing an object-oriented language with behavioural subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in LNCS, pages 60–90. Springer-Verlag, 1990.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.

[BDF+04] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[BFMW01] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass — Java with assertions. In *Workshop on Runtime Verification at CAV'01*, 2001. Published in *ENTCS*, K. Havelund and G. Rosu (eds.), 55(2), 2001.

[BJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS'01*, number 2031 in Lecture Notes in Computer Science, pages 299–312. Springer-Verlag, 2001.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2004.

[BMR95] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[BRL03] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In D. Mandrioli K. Araki, S. Gnesi, editor, *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[BSS04] Mike Barnett, David A. Naumann Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specification. In *Formal Techniques for Java-like Programs (FTfJP'2004)*, pages 11–19, May 2004. http://www.cs.ru.nl/ftfjp/2004/Purity.pdf.

[Cha06] Patrice Chalin. Towards support for non-null types and non-null-by-default in Java. In *Formal Techniques for Java-like Programs (FTfJP)*, 2006. To appear.

[CK04] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.

[CL02] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.

[CLSE05] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software:Practice and Experience*, 35(6):583–599, May 2005.

[Cok05] David R. Cok. Reasoning with specifications containing method calls in JML. *Journal of Object Technology*, 4(8):77–103, 2005.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[DM05] Á. Darvas and P. Müller. Reasoning about method calls in JML Specifications. In *Formal Techniques for Java-like Programs (FTfJP)*, 2005.

[HK00] K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In E. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221. Springer-Verlag, 2000.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa72] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[JLPS05] Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *IEEE International Conference on Software Engineering (SEFM 2005)*, pages 137–147. IEEE Computer Society, 2005.

[Jon03] Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, Department of Computer Science, January 2006. To appear in *ACM SIGSOFT Software Engineering Notes*.

[LC05] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. Draft, available from jmlspecs.org., 2005.

[LCC+05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–208, March 2005.

[LD00] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[Lea90] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90–09, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, July 1990. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[LG01] Barbara Liskov and John Guttag. *Program Development in Java*. The MIT Press, Cambridge, Mass., 2001.

[Lis88] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[LM06] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *ESOP'2006*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[LPC+06] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph R. Kiniry, and Patrice Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org, January 2006.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, October 1992.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[MHKL05] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik Luit. Cooperation-based invariants for OO languages. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS'05)*, 2005.

[MM02] Richard Mitchell and Jim McKim. *Design by Contract by Example*. Addison-Wesley, Indianapolis, IN, 2002.

[Mor94] Carroll Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[MPH00] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*, pages 63–77. Springer-Verlag, 2000.

[MPHL03] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency, Computation Practice and Experience.*, 15:117–154, 2003.

[MPHL05] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, March 2005.

[MPMU04] C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[Nau05] David A. Naumann. Observational purity and encapsulation. In *Fundamental Aspects of Software Engineering (FASE)*, 2005. Obtained from the author.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.

[PH97] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[RL00] Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 208–228, October 2000.

[Ros92] D. S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, May 1992.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Szy98] C. Szyperski. *Component Software*. Addison-Wesley, 1998.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.