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

Erik Poll[1], Patrice Chalin[2], David Cok[3], Joe Kiniry[4], and Gary T. Leavens[5]

[1] Radboud University Nijmegen, the Netherlands
[2] Concordia University, Montréal, Québec, Canada
[3] Kodak Eastman Company, R&D Laboratories, Rochester, New York, USA
[4] University College Dublin, Ireland
[5] Iowa State University, Ames, Iowa, USA

Abstract. Many state-based specification languages, including the Java Modeling Language (JML), contain at their core specification constructs familiar to most computer science and software engineering 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 the less familiar frame properties, datagroups, and ghost and model fields. These constructs help specifiers deal with potential problems related to, e.g., unexpected side effects, aliasing, class invariants, inheritance, and lack of information hiding. This tutorial focuses on 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 Tool Support

Tools useful for checking that JML annotated Java modules meet their specifications, fall into two main categories:[6]

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

These also represent two complementary forms of assertion checking, the foundations of which were laid out before the 1950's 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 80's; other early work includes Rosenblum's APP annotation langauge 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 that no violations of specifications can happen 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: e.g., 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 complimentary strengths. As compared to runtime assertion checking, static verification can give 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 clock. In doing so, we review basic concepts like method contracts and class invariants and introduce their JML notation.

[6] Of course, there are 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 example 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 `getHour()` on lines 17–19. The JML specification for this 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. In Fig. 1, 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 the assertion expressions appearing in the `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 the `ensures` clauses of `tick` (on lines 30–32). The expression `\old`(e) refers to the value of e in the method's pre-state, i.e. the state just before the method was invoked.

Preconditions and postconditions can be split over multiple `requires` and `ensures` clauses, as illustrated for the postcondition of `getSecond()` (on lines 24–25). 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 28–38 of Fig. 1, can be 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 which will be 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: i.e., either

- the clock seconds are less than 59 and the seconds are incremented by one, or
- the clock 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 appear during the development of the `TickTockClock` class. Informal comments, line the one on line 37 can be useful in remembering what remains to be formalized, or in escaping from formalism, although they do not help in verification.

```
1  //@ model import org.jmlspecs.lang.JMLDataGroup;
2  public class TickTockClock {
3    //@ public model JMLDataGroup _time_state;
4
5    //@ protected invariant 0 <= hour && hour <= 23;
6    protected int hour; //@ in _time_state;
7    //@ protected invariant 0 <= minute && minute <= 59;
8    protected int minute; //@ in _time_state;
9    //@ protected invariant 0 <= second && second <= 59;
10   protected int second; //@ in _time_state;
11
12   //@ ensures getHour() == 12  && getMinute() == 0  &&  getSecond() == 0;
13   public /*@ pure @*/ TickTockClock() {
14       hour = 12; minute = 0; second = 0;
15   }
16
17   //@ requires true;
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;
25   //@ ensures \result <= 59;
26   public /*@ pure @*/ int getSecond() { return second; }
27
28   /*@  requires   getSecond() < 59;
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*; i.e. only methods declared as `pure` can be used in assertion expressions. 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 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].

2.3 Lightweight vs. heavyweight

JML actually has two kinds of specification case: lightweight and heavyweight. The specification cases of the `tick()` method are lightweight. Examples of heavyweight specification cases are those given for 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 case 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 indented behavior of the method when it raises an exception. The latter case will be 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.

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 have to hold when a method terminates normally, and

```
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`

– exceptional postconditions, expressed by means of `signals` clauses, that have to 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 `TickTocClock` 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`. Hence, the specification in Fig. 1 rules out the generation

of any runtime exceptions, making the specification a lot stronger than it might appear at first sight.

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 re-establish it. 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 is a common misconception. There is more to the notion of invariant than the intuition above. 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();
  if (second == 60) { second = 0; minute++; }
  if (minute == 60) { minute = 0; hour++; }
  if (hour == 24)   { hour = 0; }
}
```

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 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 have to 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 are offered by Müller, Poetzsch-Heffter and Leavens [MPHL05]; alternative solutions are explored elsewhere [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 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, as in the specification of this public method we do not want to refer to private fields.

A JML `assignable` clause can be 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*. 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 (called the pre-state). The notion of *datagroup* 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 specifying how. So the specification of the method `tick()` could include

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

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

Object-oriented languages such as Java require some means for abstraction in assignable clauses. The `assignable` clause for `tick()` given above leaves a lot to be desired. Firstly, it exposes implementation details, because it mentions private 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 it can only assign to the three fields of `TickTockClock`, which is far to 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 the `tick` will have to modify these fields when the end of a day is reached; however, the specification of `tick` above will not allow these fields to be changed, as they are not listed in the `assignable` clause.

Datagroups [Lei98] provide a solution to this problem. The idea is that a datagroup is an abstract piece of an object's state, that can 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 used to specify `tick()`. This avoids exposing any private implementation details, and any subclasses of `TickTockClock` may extend the datagroup with additional fields it introduces.

Datagroups can be nested, by using the `in` clause to say that one datagroup is part of another one. The JML specifications for `java.lang.Object`declare a JML datagroup `objectState`. As this datagroup is inherited by all other classes, as a convention one can use this datagroup in any given class to describe what constitutes the 'state' of an object of that class. By that convention, the datagroup `_time_state` would be declared to belong to `objectState`.

Finally we note that although `assignable` clauses are needed when doing program verification, they are *not* needed to do runtime assertion checking, and hence are ignored by RAC tools.

4 Model fields

Model fields 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. It abstracts away from the particular concrete representation of time by using 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, though we do not cover that 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 class 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 datagroup can be viewed as a degenerate model field, namely a model field of some unit type.

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

```
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.

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 to use in its specifications, and 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 the principle of *behavioral subtyping*: any class has to meet the specifications of its supertypes. This ensures Liskov's Substitution Principle [LW94], i.e. that using an object of a subclass in a place where an object of the superclass is expected does not cause any surprises, and hence that the introduction of new subclasses does not break any existing code.

For example, consider the class `AlarmClock` in Fig. 4. Because `AlarmClock` is a subclass of `Clock`, it also inherits all the specifications of `Clock`, i.e. all invariants specified for `Clock` also apply to `AlarmClock`, and that any method in `AlarmClock` has to meet the specification for that method given in `Clock`. For example, the method `tick()`, which is overridden in `AlarmClock`, 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.

When it comes to method specifications, behavioral subtyping requires that preconditions of a method in a subclass can only be weakened, whereas postconditions of a method in a subclass can only be strengthened. One way to achieve this is to allow a subclass to give a new specification for a method—effectively overriding the one in the superclass—and then proving that (i) the precondition in the superclass implies the precondition in the subclass and that (ii) the postcondition in the subclass implies the postcondition in the superclass.

Instead, JML uses the principle of *specification inheritance* for method specifications [DL96]: any method specification in the subclass is "conjoined" with the specification, if any, for that method in the superclass, in such a way that the conditions (i) and (ii) above are automatically met.

The process of conjoining specs can be a bit subtle. This process is carried out by the tools, e.g. by the JML runtime assertion checker or ESC/Java2, but the human who reads or writes specification has to understand the consequences. If sub- and superclass give the same precondition for a method, then the conjoined specification will have the conjunction of the postconditions and the intersection of the frame properties. If different preconditions are given in the sub- and superclass it becomes trickier: the precondition of the conjoined spec will be the disjunction of the preconditions, and the postcondition of the conjoined spec is

```
1  class AlarmClock extends Clock {
2    //@ public model int _alarmTime;
3    //@ private represents _alarmTime = alarmMinute*60 + alarmHour*60*60;
4
5    //@ public ghost boolean _alarmEnabled = false; //@ in _time;
6
7    //@ private invariant 0 <= alarmHour && alarmHour <= 23;
8    private int alarmHour; //@ in _alarmTime;
9
10   //@ private invariant 0 <= alarmMinute && alarmMinute <= 59;
11   private int alarmMinute; //@ in _alarmTime;
12
13   private /*@ non_null @*/ AlarmInterface alarm;
14
15   public /*@ pure @*/ AlarmClock(/*@ non_null @*/ AlarmInterface alarm) {
16     this.alarm = alarm;
17   }
18
19   /*@ requires 0 <= hour && hour <= 23;
20     @ requires 0 <= minute && minute <= 59;
21     @ assignable _alarmTime;
22     @*/
23   public void setAlarmTime(int hour, int minute) {
24     alarmHour = hour;
25     alarmMinute = minute;
26   }
27
28   // spec inherited from superclass Clock
29   public void tick() {
30     super.tick();
31     if (getHour() == alarmHour & getMinute() == alarmMinute & getSecond() == 0) {
32          alarm.on();
33          //@ set _alarmEnabled = true;
34     }
35     if ((getHour() == alarmHour & getMinute() == alarmMinute+1 & getSecond() == 0)
36         ||
37         (getHour() == alarmHour+1 & alarmMinute == 59 & getSecond() == 0) ) {
38          alarm.off();
39          //@ set _alarmEnabled = false;
40     }
41   }
42
43 }
```

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`

slightly weaker than the conjunction of the postcondition, as each postcondition only has to apply in case the corresponding precondition did. See [DL96] for details.

6 Ghost fields

Like model fields, ghost fields are specification-only fields, so they cannot be referred to in 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 has to be given in a JML context. 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 going off or not (except maybe implicitly, in that the alarm is going off for 60 seconds after the alarm time). The state of an `AlarmClock` does therefore not 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 can be useful to add an extra boolean field to the state, which records if the associated alarm is going off. In Fig. 4, we have declared a boolean ghost field `_alarmEnabled`. Two assignments to this field are included in the method `tick()`. The assignments ensure that the field is true when the alarm enabled and false otherwise[7].

One can now try to capture the informal requirement that "if an alarm time has been set, and the alarm is enabled, then the alarm will go off for one minute at the specified alarm time", by formulating one or more suitable invariants relating the new ghost field `_alarmEnabled` to the 'real' state of the `AlarmClock`. There are many way to express such a relation, for instance using the following invariants:

[7] A subtle issue here is that `_alarmEnabled` 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 `_alarmEnabled`, the field has to be included in this datagroup.

```
invariant _alarmTime+60 <= 24*60*60 ==>
    ( _alarmEnabled
      <==> _alarmTime <= time && time <= _alarmTime+60);

invariant alarmtime+60 > 24*60*60 ==>
    ( _alarmEnabled
      <==>
      ( (_alarmTime <= time && time <= 24*60*60)
       ||
       (0 <= time && time <= (_alarmTime+60) % 24*60*60)
    ) );
```

Verification by ESC/Java2 will immediately point out that these invariants can 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 on twice in a row. Also, the alarm could go off for longer than 60 seconds, if one of these times are changed while the alarm is going of.

An improvement in the implementation would be to count down the number of seconds left until the alarm is disabled, and use this 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 second that the alarm will still be on.
  * If zero, the alarm is off. */
//@ private invariant 0 <= alarmSecondsRemaining
//@                     && alarmSecondsRemaining <= 60;

//@ private invariant (alarmSecondsRemaining > 0) == _alarmEnabled;
private int alarmSecondsRemaining = 0; //@ in _time;
...

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

```
        //@ set _alarmEnabled = true;
    }
  }
```

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

```
//@ public boolean model _alarmEnabled; //@ in _time;
//@ private represents _alarmEnabled == (alarmSecondsRemaining > 0);
```

Of course, one could choose to turn the ghost field into a real field, so that it can be used in the implementation. 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 can be assigned to, in annotations, using the special `set` keywords. A model field cannot be assigned to, but it changes value automatically whenever some of the state that it depends on changes, as laid down by the representation relation.

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. 7 shows `DigitialDisplayClock`, a subclass of `Clock`, which uses an integer array `time` of length 6 to represent time. For the correct functioning of the clock it will be important that this array is not aliased. If the array is aliased, code outside of this class could alter `time` and break the invariants for the array. Indeed, the fact that the (private) invariants depends on the array `time` already suggest that it 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 alias control (also known as encapsulation or confinement). JML includes support for the universe type system [MPHL03] as a way to specify and enforce constraints on aliasing. As is illustrated in Fig. 7, the `time` array is declared as a `rep`-field[8] 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

[8] `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   //@ private invariant time.length == 6;
6   //@ private invariant 0 <= time[0] && time[0] <= 9; // sec
7   //@ private invariant 0 <= time[1] && time[1] <= 5; // sec
8   //@ private invariant 0 <= time[2] && time[2] <= 9; // min
9   //@ private invariant 0 <= time[3] && time[3] <= 5; // min
10  //@ private invariant 0 <= time[4] && time[4] <= 9; // hr
11  //@ private invariant 0 <= time[5] && time[5] <= 2; // hr
12  //@ private invariant time[5] == 2 ==> time[4] <= 3; // hr
13  private /*@ 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 MyDigitalDisplayClock extends DigitalDisplayClock{
2
3    //@ requires time.length == 6;
4    /*@ pure @*/ public BrokenDigitalDisplayClock( /*@ non_null @*/ int[] time) {
5       this.time = time;
6    }
7
8    public /*@ pure @*/ int[] expose() { return time; }
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`.

breaks the guarantee that `time` will not be aliased outside this class. Verification with ESC/Java2 does not yet take universes into account; this is still the subject of ongoing work.

8 Conclusions

Preconditions, postconditions and invariants alone are insufficient to accurately specify object-oriented programs. This paper 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 tools 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 debate is far from settled, when it comes to agreement on the necessary and sufficient primitive constructs for the specification of mainstream object-oriented programs. 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).

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.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe 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, 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, 2004.

[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, 2003.

[CK04] David R. Cok and Joseph 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 Re-

port 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.

[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.

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

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

[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, 2000. Main reference for the JIVE verification system.

[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.

[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.