System Validation
192140122

JML Specifications
Run-time Checking
Static Checking

Lectures 5 - 7

Marieke Huisman

2010-2011

Contents

Chapter 1

Introduction

This handout describes the material for lectures 5, 6 and 7 of the Computer Science Master course System Validation (191140122). It contains the following topics:

- The JML Specification Language
- Runtime-checking of a program
- Abstract specifications
- Static checking of a program

Purpose of the System Validation course is to make you become acquainted with different formal tools and techniques that can help one to improve the quality of software applications.

In the first part of the course, different model checking approaches have been discussed. In particular, symbolic program execution has been used to check safety properties of a program. In addition, it has been discussed how an abstract model of the system can be developed. For such abstract, finite state models, both safety and liveness properties can be verified. To specify these properties, temporal logic has been used.

In the second part of the course, we will be more software-oriented. Instead of making an abstract model of the program, we will specify and verify the code directly. As a specification language, we use JML, the Java Modeling Language. During the three lectures the following topics will be discussed:

- What sort of specifications can be written using JML?

- How can you validate a JML specification during execution, *i.e.*, at run-time?
- How can you validate a JML specification statically, without running the code?

Goal of the handout is to give you support to follow the lectures, and to provide you something to fall back upon to study some particular topic again. However, this handout is by no means complete. Therefore, on BlackBoard you can find additional references to more in-depth descriptions of the different topics. We also refer the JML reference manual (see `http://jmlspecs.org` [32]), where for many of the JML language constructs a detailed description of its intended meaning is given.

This is the first year that this handout is used, and any feedback on it will be highly appreciated.

Chapter 2

Crash Course on JML

2.1 History and Background

JML, the Java Modeling Language [31], is a Design by Contract specification language for Java programs. The term Design by Contract (DbC) was introduced by Bertrand Meyer in 1986 for the Eiffel programming language [39]. DbC is a programming methodology where the behaviour of program components is described as a so-called contract. The user of a component only has to study the component's contract, and this should tell him exactly what he can expect from the component. The implementer of the component is free to choose any implementation, as long as it respects the component's contract. See also this Wikipedia page: `http://en.wikipedia.org/wiki/Design_by_contract` for a comprehensive description of the ideas behind Design by Contract.

Desigy By Contract is a popular methodology for object-oriented languages. In this case, the components are the program's classes. Contracts naturally correspond with the object-oriented paradigm to hide (or encapsulate) the internal state of an object (a class instance). Basically, a class contract describes for each method under what conditions it may be called, and what it guarantees about its result. In addition, a class contract may also describe general consistency properties of the class, *i.e.*, properties that the user always can rely upon.

Method contracts are also convenient to express the idea of behavioural subtyping [36]. In an object-oriented program, any subclass may be used wherever a superclass is expected. Behavioural subtyping expresses the idea that a subclass thus should behave as the superclass (at least, when it is used in a superclass context). Contracts can be used to ensure that a subclass is

indeed a behavioural subtype of a superclass. In particular, every method in the subclass should respect the method's contract of the superclass. And in addition, all the consistency properties of the superclass are inherited by the subclass. Notice that this same approach applies for interfaces and classes. An interface can be specified with its desired behaviour. Every class that implements this interface should be a behavioural subtype of the interface, *i.e.*, it should satisfy all the specifications of the interface.

As mentioned above, the use of Design By Contract for object-oriented languages dates back to 1986, where Bertrand Meyer introduced it for the language Eiffel [39]. The Eiffel compiler has a special option that can be used to check validity of the contract at run-time. Subsequently, the same ideas where applied to reason about other programming languages (including Modula III, C++, and Smalltalk, that were all handled in the Larch project, see `http://www.sds.lcs.mit.edu/spd/larch/`).

With the growing popularity of Java, several people decided to develop a specification language for Java. Gary Leavens – and his students at Iowa State University – used their experience from the Larch project, and started work on JML. JML is short for Java Modeling Language (the similarity to the name UML is intentional). They proposed a specification language, and simultaneously developed the JML run-time assertion checker, that could be used to validate the contracts at run-time.

At more or less the same time, Rustan Leino – and his team, then at the DEC/Compaq research centre – started working on a tool to reason statically about Java programs. Rustan Leino also had worked on specification of Modula III programs before. For their static verification tool, ESC/Java [34] they developed a specification language that was more or less a subset of the JML language that Gary Leavens proposed. But instead of developing a technique that would validate the contracts during execution, they developed a static verifier that could check whether all possible program executions respected the class's contract.

At the same time, several projects existed that targeted tool-supported verification of Java programs (for example the LOOP project [10], the Key project [9], and the Krakatoa project [38]). Quickly, people involved in these different projects more or less agreed that JML was the most appropriate as property specification language for their tools. And thus, as a result, JML became the de facto standard contract specification language for Java.

Ever since then, the community has worked on adopting a single JML language, with a single semantics – and this is still an on-going process. Over the years, JML has become a very large language, containing many different, potentially useful, specification constructs. However, because of

the language being so large, not for all constructs the semantics is actually understood and agreed upon, and moreover all tools that support JML in fact only support a subset of it. To solve this, a core subset of JML has been defined that contains the most common constructs of JML, for which the semantics is well-understood. Tools that support JML are expected to at least support this JML-core (and they are free to support any other language constructs). This core is defined in Section 2.9.1 of the JML Reference Manual [32].

During the System Validation course, we will mainly restrict ourselves to the language constructs that are in this core, as this already expressive enough to write non-trivial specifications about Java programs. However, occassionally, we will also use features from more advanced JML levels.

2.2 JML Method Contracts

Ingredients of a Method Contract So, what is exactly a method contract? A method contract consists of two things: it describes what is expected from the code that calls the method, and it provides guarantees about what the method will actually do.

The expectations on the caller are called the *precondition* of the method. Typically, these will be conditions on the method's parameters, *e.g.*, the argument should be a non-null pointer, but the precondition can also describe that the method can only be called when the object is in a particular state. In JML, every precondition expression is preceded by the keyword `requires`.

The guarantees provided by the method are called the *postcondition* of the method. They describe how the object's state is changed by the method, or what the expected return value of the method is. A method only guarantees its postcondition to hold whenever it is called in a state that respects the precondition. If it is called in a state that does not satisfy the precondition, then no guarantee is made at all. In JML, every postcondition expression is preceded by the keyword `ensures`.

JML specifications are written as special comments in the Java code, starting with `/*@` or `//@`. The `@` sign allows the JML parser to recognise that the comment contains a JML specification. The preconditions and postconditions are basically just Java expressions (of Boolean type). This is done on purpose: if the specifications are written in a language that the programmer is already familiar with, they are easier for him to write and to read.

```
package chapter2;

public interface Student {

    public static final int bachelor = 0;
    public static final int master = 1;

    /*@ pure */ public String getName();

    //@ ensures \result == bachelor || \result == master;
    /*@ pure */ public int getStatus();

    //@ ensures \result >= 0;
    /*@ pure */ public int getCredits();

    //@ ensures getName().equals(n);
    public void setName(String n);

    /*@ requires c >= 0;
        ensures getCredits() == \old(getCredits()) + c;
    */
    public void addCredits(int c);

    /*@ requires getCredits() >= 180;
        requires getStatus() == bachelor;
        ensures getCredits() == \old(getCredits());
        ensures getStatus() == master;
    */
    public void changeStatus();


}
```

Figure 2.1: First JML example specification `Student`

Example Figure 2.1 contains an example of a basic JML specification. It contains contracts for the methods in an interface `Student`, modeling a typical UT student. We discuss the different aspects of this example in full detail.

- For method `getName`, we specify that it is a `pure` method, *i.e.*, it may not have any (visible) side effects. Only pure methods may be used in specification expressions, because these should not have side effects.

- Method `getStatus` is also pure. In addition, we specify that its results may only be one of two values: `bachelor` or `master`. To denote the return value of the method, the reserved JML-keyword `\result` is used.

- For method `getCredits` we also specify that it is pure, and in addition we specify that its return value must be non-negative; a student thus never can have a negative amount of credits.

- Method `setName` is non-pure, *i.e.*, it may have side effects. Its postcondition is expressed in terms of the pure methods `getName` and `equals`: it ensures that after termination the result of `getName` is equal to the parameter `n`.

- Method `addCredits`'s precondition describes a condition on the method parameters, namely that only a positive number of credits can be added. The postcondition specifies how the credits change. Again, this postcondition is expressed in terms of a pure method, namely `getCredits`. Notice the use of the keyword `\old`. An expression `\old(`E`)` in the postcondition actually denotes the value of expression of E in the pre-state of the method. Thus the postcondition of `addCredits` expresses that the number of credits only increases: after evaluation of the method, the value of `getCredits` is equal to the old value of `getCredits`, *i.e.*, before the method was called, plus the parameter `c`.

- Method `changeStatus`'s precondition specifies that this method only may be called when the student is in a particular state, namely he has obtained a sufficient amount of credits to pass from the Bachelor status to the Master status. Moreover, the method may only be called when the student is still having a Bachelor status. The postcondition expresses that the number of credits is not changed by this operation, but the status is. Notice that the two pre-conditions

and the two-postconditions of `changeStatus` are written as separate `requires` and `ensures` clauses, respectively. Implicitly, these are assumed to be joined by conjunction, thus the specification is equivalent to the following specification:

```
/*@ requires getCredits() >= 180 &
             getStatus() == bachelor;
    ensures getCredits() == \old(getCredits()) &
            getStatus() == master;
*/
public void changeStatus();
```

Specifications and implementations Notice that the method specifications are written independently of possible implementations. Classes that implement this interface may choose different implementations, as long as it respects the specification. In the next chapters, we will see different implementations. One obvious implementation is using a field `credits` that keeps track of the number of credits earned by the student. However, an alternative implementation is to keep track of a list of courses (denoted by credits) and to compute the total number of credits as the sum of the credits of the individual courses. Later, in Section 4, we will see how this implementation can be shown to respect the specification of `Student`.

Method specifications do not always have to specify the exact behaviour of a method; they give minimal requirements that the implementation should respect. Considering the specification in Figure 2.1 again, the method specification for `changeStatus` prescribes that the credits may not be changed by this method. However, method `addCredits` is free to update the status of the student. So for example, an implementation that silently updates the status from Bachelor to Master whenever appropriate is according to the specification.

```
/*@ requires c >= 0;
    ensures getCredits() == \old(getCredits()) + c;
*/
public void addCredits(int c) {
   credits = credits + c;
   if (credits >= 180) {status = master};
}
```

Notice also that both `addCredits` and `changeStatus` would be free to change the name of the student, according to the specification, even though

we would typically not expect this to happen. A way to avoid this, is to add explicitly conditions `getName().equals(\old(getName()))` to all postconditions. Later, in Chapter 5, we will see specification constructs that can be used to explicitly disallow these unwanted changes in a more convenient way.

Default specifications You might have wondered why not all specifications in `Student` have a pre- and a postcondition. Implicitly though, they have. For every specification clause, there is a default. For pre- and postconditions this is the predicate `true`, *i.e.*, no constraints are placed on the caller of the method, or on the method's implementation.

Thus for example the specification of method `getStatus` actually is the following:

```
/*@ requires true;
    ensures status == bachelor || status == master;
*/
public int getStatus() {
   return status;
}
```

However, there is one exception to this. In JML all reference values are implicitly assumed to be non-null, except when explicitly annotated otherwise (using the keyword `nullable`). This means that the methods `getName` and `setName` have implicit pre- and postconditions about the non-nullity of the parameter and the result. Explicitly, their specifications are as follows:

```
//@ ensures \result != null;
/*@ pure */ public String getName();

/*@ requires n != null;
    ensures getName().equals(n);
*/
public void setName(String n);
```

Specification expressions Above, we have already seen that standard Java expressions can be used as predicates in the specifications. These expressions have to be side effect-free, thus for example assignments are not allowed. As also mentioned above, these predicates may contain method calls to pure methods.

In addition, JML defines several specification-specific constructs. The use of the `\result` and `\old` keywords have already been demonstrated in Figure 2.1, and the official language specification contains a few more of these. Besides the standard logical operators, such as conjunction `&`, disjunction `|` and negation `!`, also extra logical operators are allowed in JML specifications, *e.g.*, implication `==>`, and logical equivalence `<==>`. Also the standard quantifiers ∀ and ∃ are allowed in JML specifications, using keywords `\forall` and `\exists`. Using these, we can specify for example that an array argument should be sorted.

```
//@ requires (\forall int i, j;
                  0 <= i & i < j & j < a.length;
                                       a[i] <= a[j]);
public ... manipulateArray(int [] a) {...
```

The first argument is the declaration of the variable over which the quantification ranges. The optional second argument defines the range of the values for this variable, and the third argument is the actually universally quantified predicate. Note that an alternative way to phrase this is the following:

```
//@ requires (\forall int i, j;
                  0 <= i & i < j & i < a.length ==>
                                          a[i] <= a[j]);
public ... manipulateArray(int [] a) {...
```

The official JML syntax also allows other quantified expressions, such as `\sum` and `\num_of`, but unfortunately most tools do not support these yet.

A Note on Purity Above, we have said that a method should be pure if it is to be used in a method specification, and purity was defined as having no visible side effects. No visible side effects means that the state that was allocated before the method call may not be changed. Thus, this does not exclude that a method creates a new object and initialises that. In fact, even changing a location and then restoring it to its old value before the method is finished technically speaking does not make a method non-pure.

To complicate matters even further, some methods exist that are technically speaking not pure, but from a specification point-of-view may be considered to be so. Consider for an example the function that computes a hashcode. The first time this function is called on an object, a field of the object will be written, so that the next calls can be evaluated by looking up this field. Because of this, different variations of purity and observational

purity exist in the literature, see *e.g.*, [5, 19, 18] for more information. For the scope of this course, it is sufficient to define purity simply as not having any side effects.

Behaviours An important question is when a method specification is actually verified, and in particular if a method does not terminate, does it then satisfy its specification?

The specifications as we have seen here are so-called *lightweight specifications*, and it specifies a *partial correctness* condition. If method `m` is specified as follows:

```
/*@ requires P;
    ensures Q;
*/
public ... m(...) { ...
```

this means the following: *if* method `m` is executed in a pre-state where `P` holds, *and if* execution of method `m` from this pre-state terminates, then in the post-state the postcondition `Q` holds.

Thus, if method `m` never terminates, it vacuously respects any lightweight specification.

If one explicitly wishes to specify that a method has to terminate, this can be done by adding a so-called `diverges` clause. A diverges clauses specifies under which conditions a method may not terminate. If no diverges clause is specified, this is assumed to be `not_specified` for a lightweight specification.

Consider the following specification:

```
/*@ requires P;
    ensures Q;
    diverges false;
*/
public ... m(...) { ...
```

This specifies that method `m` has to terminate - it only may diverge if `false` holds, which is never the case. Notice however that to express that a method must always terminate, JML provides more convenient ways to express this, as discussed shortly below. Nevertheless, diverges clauses can be useful, in particular if one wishes to expresses that for certain parameters a method might not terminate:

```
/*@ requires P;
    ensures Q;
```

```
    diverges x < 0;
*/
public ... m(int x) { ...
```

To express directly that a method must terminate, JML also provides heavyweight specifications. These are preceded by a keyword `...behavior`. Standard behavior specifications, preceded by the keyword `behavior` also specify partial correctness specifications, but their default diverges clause is `false`. Thus, unless specified explicitly otherwise, a behavior specification specifies that a method must terminate. However, this does not exclude that a method may terminate because of an exception. If we also want to exclude this case, then a `normal_behavior` specification is to be used: this states that method execution has to terminate in a normal state, and in the post-state the postcondition has to hold[1].

Thus, to summarise, consider the following specifications:

```
/*@ requires P1;
    ensures Q1;
*/
public ... m1(...) { ...
}

/*@ behavior
      requires P2;
      ensures Q2;
*/
public ... m2(...) { ...
}

/*@ normal_behavior
      requires P3;
      ensures Q3;
*/
public ... m3(...) { ...
}
```

these specifications state the following. If the method `m`i (for $i = 1, 2, 3$) is executed in a pre-state where precondition `P`i holds, then:

[1] A method is said to terminate normally if either it reached the end of its body, in a normal state, or it terminated because of a `return` instruction.

- `m1` may not terminate, and if `m1` terminates, then postcondition `Q1` holds;
- `m2` has to terminate, either normally or with an exception, and if `m2` terminates normally, then postcondition `Q2` holds; and
- `m3` has to terminate normally, and in the post-state postcondition `Q3` has to hold.

Specifications for constructors Constructors can be considered as special methods. In the pre-state of a constructor, the object does not yet exist. Thus a precondition of a constructor can only put constraints on the constructor parameters, it cannot put require anything about the internal state of the object – as the object does not exist yet when the constructor is called. However, the postcondition of the constructor can specify constraints on the state of the object. Typically, it will relate the object state to the constructor's parameters.

Suppose we have a class `CStudent`, implementing the `Student` interface. It could have the following constructor:

```
/*@ requires c >= 0;
    ensures getCredits() == c;
    ensures getStatus() == bachelor;
    ensures getName() = n;
*/
CStudent (int c, String n) {
   credits = c;
   name = n;
   status = bachelor;
}
```

Thus, to repeat, it would be incorrect to specify *e.g.*, `requires getCredits() >= 0;` or `requires getStatus() == bachelor` – these specifications are meaningless at the moment that the constructor is invoked.

Defensive versus offensive method implementations A last important point about method contracts is that they can be used to avoid defensive programming. Consider the specification of method `addCredits` in Figure 2.1.

This method assumes that its argument is non-negative, and otherwise it is not going to function directly. When one uses a defensive programming

style, then one would first test the value of the argument and throw an exception if this was negative. This clutters up the code, and in many cases it is not necessary. Instead, using specifications, one can use an "offensive" coding style. The specification states what the method requires from its caller. It only guarantees the function correctly the caller also fulfills its part of the contract. When validating the application, one checks that every call of this method is indeed within the bounds of its specification, and thus the explicit test in the code is not necessary. Thus, making good use of specifications can avoid adding many parameter checks in the code. Such checks are only necessary when the parameters cannot be controlled – for example, because they are given via an external user.

2.3 JML Class Specifications

Consider again the specification of `Student` in Figure 2.1. If we look carefully at the specifications and the description that we give about the student's credits, we notice that actually we implicitly assume some properties about the value of `getCredits` that hold throughout. For example, we wrote above:

> "a student thus never can have a negative amount of credits."

and also

> "the number of credits only increases"

But if we would like to make explicit that we assume that these properties always hold, we would have to add this to *all* the specifications in `Student`, and thus in particular also to all methods that do not relate at all to the number of credits. Thus for example, we would get the following specification:

```
/*@ requires getCredits() >= 0;
    ensures \result == bachelor || \result == master;
    ensures getCredits() >= 0;
*/
/*@ pure */ public int getStatus();
```

Clearly, this is not desired, because specifications would get very large, and besides describing the intended behaviour of that particular method, they also describe properties over the lifetime of the object.

Therefore, JML provides also class-level specifications, such as invariants, constraints and initially clauses. These specify properties over the

internal state of an object, and how the state can evolve during the object's lifetime.

Invariants An object invariant[2] is a predicate over the object state that holds in all *visible* states of an object. A visible state of an object is defined to be any state in which a method call to the object either starts or terminates. Thus, an invariant I is implicitly added as a precondition and a postcondition to every method in the class. In addition, also the poststates of the constructor are visible states, thus any constructor has to ensure that the invariant is established.

Figure 2.2 shows three possible invariants that can be added to interface `Student`. These specify that credits are never non-negative; a student's status is always either bachelor or master, and nothing else; and if a student's status is master, he or she has earned more than 180 credits.

Of course, instead of specifying invariants, one can add these specifications to all pre- and postconditions explicitly. However, this means that if you add a method to a class, you have to remember to add these pre- and postconditions yourself. Moreover, invariants are also inherited by subclasses (and by implementations of interfaces). Thus any method that overrides a method from a superclass still has to respect the invariants. And any method that one adds in the subclass, still has to respect the invariants from the superclass. This leads to a very nice separation of concerns.

An important point to realise is that invariants have to hold only in all *visible object states*, *i.e.*, in all states in which a method is called or terminates. Thus, inside the method, the invariant may be temporarily broken. So for example, the following possible implementation of `addCredits` is correct, even though it breaks the invariant inside the method: if `credits + c` is sufficiently high, the status is changed to master. After this assignment the invariant does not hold, but because of the next assignment, the invariant is re-established before the method terminates.

```
/*@ requires c >= 0;
    ensures getCredits() == \old(getCredits()) + c;
*/
public void addCredits(int c) {
   if (credits + c>= 180) {status = master;} // invariant broken!
   credits = credits + c;
}
```

[2]Not to be confused with loop invariants, as discussed in Programmeren 1. These will be discussed in Chapter 5.5.

```
package chapter2;

interface StudentWithInv {

    public static final int bachelor = 0;
    public static final int master = 1;

    //@ invariant getCredits() >= 0;
    //@ invariant getStatus() == bachelor ||
    //@           getStatus() == master;
    //@ invariant getStatus() == master ==>
    //@              getCredits() >= 180;


    //@ initially getCredits() == 0;
    //@ initially getStatus() == bachelor;

    //@ constraint getCredits() >= \old(getCredits());
    //@ constraint \old(getStatus()) == master ==>
    //@                 getStatus() == master;
    //@ constraint \old(getName()) == getName();


    /*@ pure */ public String getName();

    /*@ pure */ public int getStatus();

    /*@ pure */ public int getCredits();

    //@ ensures getName().equals(n);
    public void setName(String n);

   /*@ requires c >= 0;
       ensures getCredits() == \old(getCredits()) + c;
    */
    public void addCredits(int c);

    /*@ requires getCredits() >= 180;
        requires getStatus() == bachelor;
        ensures getCredits() == \old(getCredits());
        ensures getStatus() == master;
    */
    public void changeStatus();


}
```

Figure 2.2: Interface `Student` with class-level specifications

```
package chapter2;

interface CallBack {

    //@ invariant getX() > 0;
    //@ invariant getY() > 0;

    /*@ pure */ public int getX();
    /*@ pure */ public int getY();

    //@ ensures getX() == x;
    public void setX(int x);

    //@ ensures getY() == y;
    public void setY(int y);

    //@ ensures \result == getX() % getY();
    public int remainder();

    public int longComputation();

}
```

Figure 2.3: Interface `CallBack`

However, if a method calls another method on the same object, it has to ensure that the invariant holds before this callback. Why this is necessary, is best explained with an example. Consider the interface `CallBack` in Figure 2.3.

Typically, correctness of the method `remainder` crucially depends on the value of `getY` being greater than 0. Suppose we have an implementation of the `CallBack` interface, where the method `longComputation` is sketched as follows.

```
public int longComputation(){
   ...
   if (getY() ....) {
      setY(0); // invariant broken
   }
```

```
    ...
    int r = remainder(); // callback
    ...
    setY(r + 1) // invariant re-established
    ...
    return ...
}
```

Naively, one could think that the fact that the invariant about `getY` is broken inside this method, is harmless, because the invariant is re-established by the `setY(r + 1)` statement. However, the call to the method `remainder` is a callback, and the invariant should hold at this point. In fact, correct functioning of this method call depends on the invariant holding. The invariant implicitly is part of `remainder`'s precondition. If the invariant does not hold at the point of the callback, this means that `remainder` is called outside its precondition, and no assumption can be made about its result as well.

There is a way to avoid the requirement that the invariant has to hold upon callback: this is by specifying that a method is a `helper` method. Such methods cannot depend on the invariant to hold, and they do not guarantee that the invariant will hold afterwards. Typically, only private methods should be specified as helper methods, and one does not want that any other object can directly invoke a helper method.

Defining a precise semantics for invariants is still an active area of research, see *e.g.* [44, 35, 42, 3]. An alternative approach, that is used in the Spec# framework, is to explicitly unpack and pack invariants. An invariant may only be broken if it has been explicitly unpacked. When the invariant is re-established, it has to be explicitly be packed again. Every method can then specify explicitly whether it assumes invariants to hold, *i.e.*, to be packed, or not. This approach is sometimes referred to as the Boogie methodology [2].

Finally, it is important to realise that the notion of object invariant that we discussed here only makes sense in a sequential setting. In a multithreaded setting, there always may be another thread accessing the object simultaneously, and cannot talk about visible state semantics anymore. Instead, in a multithreaded setting, one sometimes specifies so-called *strong invariants* that may never be broken.

Initially Clauses Sometimes, one explicitly wishes to specify the conditions that are satisfied by an object upon creation. Each (non-helper)

constructor of the object has to establish the predicate specified by the initially clause. Figure 2.2 shows some possible initially clauses for the `Student` interface.

Again, it would be possible to specify this property as a postcondition of all constructors, instead of as a single initially clause. But in this way, we ensure that also subclasses respect the initially clause, and that any additional constructor has to respect it.

Constraints Invariants as we discussed above define a predicate that every (visible) state of the object should respect. However, sometimes one also wishes to specify how an object may evolve over time, *i.e.*, the relationship that exists between the pre-state and the post-state of a method call. For this, history constraints (usually constraints for short) have been introduced [36].

Figure 2.2 defines several constraints for the `Student` interface. The first constraint specifies that the amount of credits can never decrease. The second constraint specifies that if a student has obtained the master status, he will remain a master student, and cannot be downgraded to a bachelor student again. Finally, the third constraint specifies that a student's name can never change.

When specifying constraints, it is important that they should be "*non-strict*", *i.e.*, it should be possible to respect a constraint without actually changing the state that is specified by the constraint. In particular, any pure method should be able to respect the constraint. Therefore, one should not specify the following strict constraint:

```
constraint \old(getCredits()) < getCredits();
```

as it is impossible to respect this constraint with a pure method.

Constraints are implicit postconditions, but just as invariants and initially clauses, they have the advantage that they are inherited, and immediately are required to hold for any additional methods.

As for invariants, constraints are specified in terms of visible states. In particular, a constraint is a relation that has to be satisfied by any two consecutive visible states.

Consider the following possible implementation of `addCredits`.

```
/*@ requires c >= 0;
    ensures getCredits() == \old(getCredits()) + c;
*/
// pre-state
```

```
public void addCredits(int c) {
   credits = credits + c;
   if (credits >= 180) {
      // call-state changeStatus
      changeStatus();
      // return-state changeStatus
   }
} // post-state
```

To show that the constraint is respected, it has to hold for the following visible state pairs:

- (pre-state, call-state `changeStatus`)
- (call-state `changeStatus`, return-state `changeStatus`)
- (return-state `changeStatus`, post-state)

Again, in a multithreaded setting, the notion of constraint becomes unclearer. However, a constraint such as that `getName` returns a constant value could still be meaningful also in a multithreaded setting (except that the number of possible visible state pairs that have to be considered might grow exponentially).

Variable declarations So far, the specifications that we have seen have not specified anything about the values of an object's instance variables. Typically, these are declared private, and private elements should not be used in (public) specifications. The specifications should only be written in terms of public elements that are visible outside the class[3].

Therefore, in our examples, the pure get-methods were used to specify how the internal state of the object was changed. However, sometimes this is not possible, or not convenient. As an alternative, one can specify that a variable should be `spec_public`. This means that the variable *at specification-level* has public visibility, and in particular that the specifications can be written in terms of these variables.

If we specify the instance variables of class `CStudent` to be `spec_public`, then its constructor can also be specified as follows.

[3]In fact, the story is more subtle: the standard Java access modifiers can be used to control visibility of the specifications, see the JML reference manual [33]. However, for this course it is sufficient to consider specifications to be publicly visible.

```
class CStudent implements Student {

    /*@ spec_public */ private String name;
    /*@ spec_public */ private int credits;
    /*@ spec_public */ private int status;
    ....

    /*@ requires c >= 0;
        ensures credits == c;
        ensures status == bachelor;
        ensures name = n;
    */
    CStudent (int c, String n) {
    credits = c;
    name = n;
    status = bachelor;
    }
}
```

In Chapter 4, we will see that `spec_public` variables are actually a special instance of model variables that directly represent the underlying private variables.

Static class specifications For all class-level specification constructs, static variants exists. For example, an invariant might restrict the value of a static variable, or a constraint might restrict the evolution of a static variable. Since instance methods might change static variables, static invariants and constraints have to be respected by instance methods. In contrast, invariants and constraints that only restrict the instance variables of a method cannot be invalidated by a static method – and thus this does not have to be checked explicitly.

2.4 Specifying Exceptional Behaviour

So far, we have only considered normal termination of methods. But in some cases, exceptions cannot be avoided. Therefore JML also allows one to specify explicitly under what conditions an exception may occur.

The `signals` and `signals_only` clauses are introduced to do exactly this. The signals clause is part of a method specification. Its syntax is `signals (E e) Predicate`, and it has the following meaning: if the method

```
package chapter2;

class Average {

    /*@ spec_public */ private Student[] sl;

    /*@ signals_only ArithmeticException;
        signals (ArithmeticException e) sl.length == 0;
    */
    public int averageCredits() {
    int sum = 0;
    for (int i = 0; i < sl.length; i++) {
        sum = sum + sl[i].getCredits();
    };
    return sum/sl.length;
    }
}
```

Figure 2.4: Class `Average`

terminates because of an exception that is an instance of class `E` – thus, its dynamic type may also be a subtype of `E` – , then the `Predicate` has to hold. The variable name `e` can be used to refer to the exception in the predicate. The `signals_only` clause is also part of the method specification. Its syntax is `signals_only E1, E2, .., En`, meaning that if the method terminates because of an exception, the dynamic type of the exception has to be a subclass of `E1`, `E2`, ..., or `En`.

Consider for example class `Average` in Figure 2.4. The specification of method `averageCredits` states that the method may only terminate with an `ArithmeticException` – and thus, it will not throw an `ArrayIndexOutOfBoundsException`, and if this exception occurs, then this is caused by the fact that the length of `sl` is 0.

Notice that it is incorrect to use an `ensures` clauses, instead of `signals` clause: an `ensures` clause specifies a normal postcondition, that only holds upon normal termination of the method.

Above, in Section 2.2 we have seen that a method can have a so-called `normal_behavior` specification. Implicitly, this says that the method has to terminate normally. Similarly, JML also has an `exceptional_behavior` method specification. This specifies that the method has to terminate, be-

cause of an exception. As mentioned above, a `behavior` specification only enforces that a method terminates, but it does not exclude exceptional termination. Thus a `behavior` specification may well contain a `signals` or `signals_only` clause, whereas a normal behaviour specification may not contain these, and an exceptional behaviour specification may not contain an `ensures` clause.

Such exceptional behaviour specifications are mainly used when several method specifications are combined. A single method can be specified with several method specifications, joined with `also`. This should be interpreted as a conjunction of method specifications. Consider for example the more detailed specification for `averageCredits` in Figure 2.5.

This states that if `sl.length > 0`, *i.e.*, there are students in the list, then the method terminates and the result is the average value of the credits obtained by these students. If `sl.length == 0` then the method will terminate exceptionally, with a `ArithmeticException`. In this example, the two preconditions together cover the complete state space for the value of `sl.length`. If `sl.length` could be less than 0, the method's behaviour would not be specified.

Finally, it is important to realise that invariants and constraints also should hold when a method terminates exceptionally. This might seem strange at first: something goes wrong during the execution, so why would it be necessary that the object stays in a good state. But in many cases, the object can recover from the exception, and execution on it can continue. But this means that it is necessary that also when an exception occurs, the object stays in a "well-defined" state, *i.e.*, a state in which the invariants hold, and that evolves according to the constraints.

2.5 Desugaring Multiple Method Specifications

Combining different method specifications with an `also` is convenient to obtain readable specifications. However, to develop tool support, it is better to have a single specification per method.

Therefore, a method specification desuguaring procedure has been defined [45] that transforms specifications in such a way that every method is annotated with exactly one method specification. This desugaring procedure also incorporates invariants, initially clauses and constraints into method specifications, where appropriate.

For the purpose of System Validation, it is not necessary to understand all the details of the transformation. However, to understand the feedback

```
package chapter2;

class Average {

    /*@ spec_public */ private Student[] sl;

    /*@ normal_behavior
          requires sl.length > 0;
          ensures \result ==
             (\sum int i; 0 <= i && i < sl.length;
                          sl[i].getCredits())/sl.length;
        also
        exceptional_behavior
          requires sl.length == 0;
          signals_only ArithmeticException;
          signals (ArithmeticException e) true;
    */
    public int averageCredits() {
    int sum = 0;
    for (int i = 0; i < sl.length; i++) {
        sum = sum + sl[i].getCredits();
    };
    return sum/sl.length;
    }
}
```

Figure 2.5: Class `Average`

from the tools that validate the JML specifications, it is important to have some idea how this procedure works. For a detailed description, we refer to the paper that describes the desugaring procedure in full detail [45].

The main steps are as follows:

- Consider the method specifications that are combined with `also`. The complete precondition for the method, *i.e.*, the precondition for which the methods behaviour is described, is the disjunction of the individual preconditions.
- The (normal or exceptional) postconditions only have to hold if the appropriate precondition was true in the pre-state. Therefore, every

postcondition is preceded by an implication about the precondition in the pre-state (using `\old`).

- The requirement that the method terminates (either normally or exceptionally) is expressed using a diverges clause.
- Invariants are added as pre- and postconditions.
- Constraints are added as postconditions.
- Invariants and initially clauses are added as postconditions to constructors.

For method `averageCredits` the desugaring procedure returns the following single method specification (where some obvious simplifications can be applied):

```
/*@ behavior
      requires sl.length > 0 || sl.length == 0;
      ensures \old(sl.length > 0) ==>
           \result ==
           (\sum int i; 0 <= i && i < sl.length;
                        sl[i].getCredits())/sl.length;
      signals_only ArithmeticException;
      signals (ArithmeticException e) \old(sl.length == 0);
      diverges sl.length > 0 ==> false;
      diverges sl.length == 0 ==> false;
    */
```

2.6 Inheritance of Method Specifications

Above, we have already mentioned that subclasses inherit class-level specifications (invariants, initially clauses and constraints). Method specifications are also inherited. Concretely, this means the following:

- every class that implements an interface has to respect the specifications of the interface; and
- every class that extends a another class has to respect the specifications of these class.

This means in particular that every method that overrides a method from a superclass should still respect the method specification from the superclass. Any additional specification of the subclass is implicitly combined (with `also`) with the specifications from the superclass[4].

This makes the subclass a *behavioural subtype* of the superclass [36], *i.e.*, in any context where an instance of the superclass is expected, an instance of the subclass can be safely used, and when you look at it from the superclass perspective, you cannot observe any difference in behaviour.

This idea is crucial for the correctness of object-oriented programs. You can specify the behaviour of a class in an abstract way. For example, if you have an array of students, as in class `Average`, you know that the concrete instances that may be stored in the array may have different implementations, but you know that they all implement the methods specified in the interface `Student`, respecting the behaviour that is specified for this interface.

[4] And the desugaring procedure will explicitly repeat the specifications from the superclass.

Chapter 3

Run-time checking of JML annotations

Now that we have seen how we can specify software using JML, the next question is how to validate that a program indeed respects its specification. We will look at different techniques for this. This chapter discusses how the program can be instrumented in such a way that during execution, validity of the specification can be checked. The next chapter discusses how this validity check can be done at compile time, without actually executing the program.

When work on JML started, its initial goal was to support run-time checking of specifications. The idea of run-time checking is that every precondition and postcondition is checked by simply executing the predicate and testing whether the outcome of this evaluation is `true`. Execution of the precondition should be done at an appropriate point in the code:

- a precondition should be evaluated before the method body is executed; and
- after the method body has terminated, the postcondition should be evaluated, before control is given back to the caller.

It is important to realise that to make this approach work in practice, one needs a desugaring procedure that can translate a bunch of method specifications into a single method specification. In this way, for every method a single precondition and a single postcondition test can be implemented.

This approach is often advocated as a good way to test your program. For example, in the text book on object-oriented programming by Niño and Hosch [43] (used in Programmeren 1 and 2), it is shown how precondition

specifications can be translated systematically into assert-statement at the beginning of a method body.

One of the examples that we will use in this chapter is the class `CStudent`. This implements the interface `Student` introduced in the previous chapter by defining an explicit `credit` field. In addition, it also contains several extra methods, discussed below. The implementation of class `CStudent` is given in Figures 3.1, 3.2. Notice that we do not have to repeat the specifications that are inherited from `Student`. A remark on notation: {| ... |} is used for nested specifications: the two specifications of `activityBonus` conjoined with `also` have a common precondition `0 <= bonus && bonus <= 5 && active;`.

Consider the method `addCredits` with the following (inherited) specification.

```
/*@ requires c >= 0;
    ensures getCredits() == \old(getCredits()) + c;
*/
```

Using the approach as advocated by *e.g.*, Niño and Hosch, a programmer would have to transform this method implementation manually into the following.

```
public void addCredits(int c) {
   assert c >= 0;
   credits = credits + c;
}
```

Niño and Hosch argue that one does not have to test for the postcondition, as this is the programmer's responsibility. In their view, a programmer would have to use other means to ensure that their own implementation is correct (such as static checking for example, as discussed in the next chapter). However, one can never be sure that the method is called under the right conditions, thus it is important to test for the precondition. If one would wish to test for the postcondition as well, this could be done by adding an additional `assert` at the end of the method body.

However, inserting all these checks manually is cumbersome and error prone. Besides the amount of work, there are are also more important problems, such as:

- a method might have multiple exit points, and the postcondition has to be checked at every exit point;

```
package chapter3;

import chapter2.Student;

class CopyOfCStudent implements Student {

    private String name;
    private int credits;
    private int status;
    /*@ spec_public */ private boolean active;

    /*@ requires c >= 0;
        ensures getCredits() == c;
        ensures getStatus() == bachelor;
        ensures getName() == n;
    */
    CopyOfCStudent (int c, String n) {
        credits = c;
        name = n;
        status = bachelor;
    }

    public String getName() {
    return name;
    }

    public int getStatus() {
    return status;
    }

    /*@ pure */ public int getCredits() {
    return credits;
    }

    public void setName(String n) {
    name = n;
    }
```

Continued in Fig. 3.2

Figure 3.1: Class `CStudent` with explicit `credit` field (1/2)

Continued from Fig. 3.1

```
    public void becomeActive() {
    active = true;
    }

    public void addCredits(int c) {
    credits = credits + c;
    }

    public void changeStatus() {
    status = master;
    }

    /*@ requires 0 <= bonus && bonus <= 5 && active;
        {|requires getStatus() == bachelor;
          ensures getCredits() == (\old(getCredits()) +
             bonus > 180 ? 180 : \old(getCredits()) +
             bonus);
          also
          requires getStatus() == master;
          ensures getCredits() == \old(getCredits() +
             bonus);
        |}
    */
    public void activityBonus(int bonus) {
        if (active) { addCredits(bonus);}
    }


}

class ExecuteCStudent {

    public static void main (String [] args) {
    CopyOfCStudent s = new CopyOfCStudent(0, "marieke");
    s.becomeActive();
    s.addCredits(178);
    s.activityBonus(5);
    System.out.println(s.getCredits());
    System.out.println(s.getStatus());
    }
}
```

Figure 3.2: Class `CStudent` with explicit `credit` field (2/2)

- a method might terminate exceptionally, to evaluate the (exceptional) postcondition, the state temporarily has to be brought back to normal again;
- specification-only expressions cannot be used in a Java assert, and encoding them is often quite complex;
- values of expressions in the pre-state have to be stored explicitly; and
- class-level specifications would give rise to asserts in many different places.

3.1 Systematically Inserting Run-Time Checks

Therefore, it is a logical idea to have tool support for doing this. Bertrand Meyer was the first to introduce this idea as an integral part of a programming language. To support the Design by Contract approach [39], the Eiffel compiler came with a special option to insert these run-time checks in the code. Also JML has a special tool that inserts these run-time checks [15]. Initially, the run-time checker was implemented by inserting explicit checks in the source code. Later, this was adapted, and now explicit tests are added to the bytecode.

A run-time checker has to satisfy three important requirements.

- Run-time checking should be *transparent*: if there are no annotation violations, then execution with or without run-time checking enabled should be identical[1]
- Run-time checking should *isolate* the source of the problem. An annotation violation should be reported when it occurs, and it should relate back to the point where the problem occurred, so that the user can actually trace the problem.
- Run-time checking should be *trustworthy*: if an annotation violation is signalled, it should indeed be a real annotation violation.

Adding run-time checking for a large part is engineering. To ensure that in case of exceptional termination the appropriate checks are executed, the code is wrapped in the bytecode equivalents of a `try-catch-finally` statement. In the catch block, the appropriate exceptional postconditions (including invariants and constraints) are tested, and if these succeed, then the appropriate exception is rethrown.

[1] Apart from performance, of course.

```
package chapter3;

class ExecuteCStudent2 {

    public static void main (String [] args) {
    CStudent s = new CStudent(0, "marieke");
    s.becomeActive();
    s.addCredits(178);
    s.activityBonus(5);
    System.out.println(s.getCredits());
    System.out.println(s.getStatus());
    }
}
```

Figure 3.3: Method `main` to execute class `CStudent`

3.2 Executing a Run-Time Checker

To illustrate the use of a run-time checker, we consider the method `activityBonus` of class `CStudent`. If a student is active (for example as a member of the Interactief board), the university can provide him a maximum of 5 bonuscredits for this. However, as specified, a Bachelor student can never obtain Master credits with this bonus, *i.e.*, the new credits can never be more than 180. Unfortunately, the programmer did not correctly implement this rule.

We show how to use the run-time checker to find this problem. As mentioned above, we have to make an executable program. Thus, we write a `main` method, as displayed in Figure 3.3. This first creates a new student object. Then we have to bring the object into a state where the annotation violation will be detected. If we just call `activityBonus` once with argument 5, then the annotation violation will not be detected. Instead, we first make the student active by executing `becomeActive`, and then we call `addCredits`, where there is no bound on the credits that can be added. We use this to give the student 178 credits. Then we call `activityBonus`.

If we now execute the program with the run-time checker, this will produce the following error message (with a few linebreaks added for readability).

```
Exception in thread "main" org.jmlspecs.jmlrac.runtime.
```

```
JMLInternalNormalPostconditionError:
by method CStudent.activityBonus
    at chapter3.CStudent.activityBonus
           (ExecuteCStudent.java:1895)
    at chapter3.ExecuteCStudent.internal$main
           (ExecuteCStudent.java:2116)
    at chapter3.ExecuteCStudent.main
           (ExecuteCStudent.java:2390)
```

When using the run-time checker to detect annotation violations, it is important that many different corner cases are checked. Just as with more traditional testing techniques, the corner cases are the most likely to cause problems. Also, it is important to try to get a good code coverage. By testing executions that cover all possible execution paths, one can increase the confidence that the implementation respects the specifications. The JML annotations typically guide one in developing these test cases. This idea is also exploited in the tool JMLUnit [48]; an extension of JUnit that generates test cases based on the JML annotations.

3.3 Other run-time checkers

Run-time checking is a commonly used validation technique. In this course, we focus on run-time checking of annotations. Some tools have been developed that apply similar approaches (*e.g.*, Jass for Java applications [30, 6], and Code Contracts for Microsoft's C# [16]). Also aspect-oriented and other compositional approaches are used to introduce run-time checks into code by weaving (see *e.g.*, [1, 46]).

Much work on run-time also exists for temporal properties. As a spin-off of the JPF project, a project called Java Path Explorer developed run-time checking techniques for safety and bounded liveness properties [22].

Finally, another active branch of research on run-time checking is the checking of dedicated properties. If you want to check a single property on your program, *e.g.*, the program does not use too much resources, confidential information does not flow to publicly visible variables, or response time of certain actions is not too long, a possible way to guarantee this is to use run-time checking and to break off program execution if the desired property is (or might be) violated.

3.4 Monitoring of safety and security properties

Many common safety and security properties can be expressed as a simple automaton that describe the legal behaviours, not violating security. Examples of such properties are:

- at most one file can be opened at the same time;
- every send message should be preceded by a read message;
- protected data may only be accessed by un authorised user;
- an SMS should only be send after asking authorisation; and
- if a pincode is entered incorrectly three times consecutively, a paycard gets blocked.

A commonly used way to guarantee that an application respects such a property is *monitoring of the automaton*. The idea, originally introduced by Schneider [47], is that the application is run in parallel with the automaton that encodes the desired property. The automaton describes the abstract state of the program. Upon certain dedicated events in the program, typically calls and returns to methods, the automaton makes a transition to a new state. If the automaton reaches a special error state (or is unable to make a transition, depending on how the property is precisely modeled), this means that the property that is being monitored is violated. Thus, to avoid any problems, the application has to be stopped immediately. Notice that this approach in this basic form only works for safety properties. However, in the literature one can find approaches for monitoring liveness properties, assuming some bounds on when the events eventually have to happen, see *e.g.*, [22].

With JML, a similar monitoring approach can be achieved. A safety property can be encoded as a collection of JML annotations, and violation of the JML annotations during program execution ensure that the application will terminate.

During this course, we will study how we can encode a security property manually. In the literature, also some more systematic translations are developed, see *e.g.*, the AutoJML tool [25] and the work of Huisman and Tamalet [27], who define a formal translation and prove this correct.

Consider the class `Card` in Figure 3.4. It has a method `initializeCode` that be used to initialize the pincode. After initializing the card, before a payment can be made, a pincode has to be entered. The `enterPincode`

```
package chapter3;

interface Card {

    void initialiseCard();

    boolean enterPin(int pin);

    void makePayment();

}
```

Figure 3.4: Class `Card`

method returns a boolean, denoting whether this was the correct pincode. If there are three consecutive wrong attempts to enter the pincode, the card gets blocked. Properties such as this one are often called life cycle properties. They are another typical example of the properties that can be encoded with this approach.

To encode this, we add *ghost* variables to the interface. These are specification-only variables that can be updated in the specification, via a special *set* annotation. Ghost variables are typically used to encode some control information about the state. First, we add constants (public static final) to encode the different states. We also define a special state `BLOCKED` that we actually never want to reach. Then we add a ghost variable `state` that is used to keep track of the card state. It is initialised to `FRESH`.

We add an invariant that states which values we want `state` to have. Notice that here we explicitly do not allow the state to be `BLOCKED`: if the card enters a blocked state, an annotation violation will be signalled.

Next, we add a constraint that specifies how state transitions can happen: from the `FRESH` state, one only can move to a `READY_FOR_USE` state, from `READY_FOR_USE`, one can move to `READY_FOR_PAYMENT` and `BLOCKED`, and when a card is `BLOCKED`, it will stay blocked forever. Notice that the constraint also allows in all cases to not change state: this is the so-called *non-strictness* condition on constraints: it should be possible to call a pure method, and to still respect the constraint (even though in this very simple example, it is not strictly necessary).

Further, we add an extra invariant, stating that the state can only become blocked if the number of wrong pin attempts is 3. This actually could

have been part of the constraint of as well, but we felt it was clearer to specify this separately.

Finally, to make the implementation respect the specification, state transitions have to be inserted at appropriate places. These correspnd to the events that make the automaton transition in the monitoring approach. The state transitions are added here as *statement annotations*, *i.e.*, annotations that are part of the code, instead of the documentation, and that only serve to show correctness of the class documentation. The statement annotations here start with the `set` keyword. Basically, they describe an assignment to a ghost variable. (It is also possible to write more complex statements, using *e.g.*, `if` – provided the statement only has a side effect on the ghost variables.

The complete result of the annotation process is depicted in Figures 3.5, 3.6.

An advantage of this approach over the traditional monitoring approach is that the JML annotations can be validated in different ways. In Chapter 5 we will discuss how annotations can be validated without actually executing the code. Monitoring often has a significant performance overhead. If we can use static methods to guarantee that certain parts of the code never violate the security or safety properties, then the run-time checks can be avoided in these places – thus reducing execution overhead.

```
package chapter3;

class CardImpl implements Card {

    /*@ ghost public static final int FRESH = 0;
        ghost public static final int READY_FOR_USE = 1;
        ghost public static final int READY_FOR_PAYMENT = 2;
        ghost public static final int BLOCKED = 3;
    ghost public int state;

        ghost public int attempts;
     */

    /*@ invariant state == FRESH ||
                   state == READY_FOR_USE ||
          state == READY_FOR_PAYMENT;
    */

    /*@ initially state == FRESH;
     */

    /*@ constraint
          (\old(state) == FRESH ==> state == FRESH ||
                                    state == READY_FOR_USE) &
          (\old(state) == READY_FOR_USE ==>
                state == READY_FOR_USE ||
                state == READY_FOR_PAYMENT ||
                state == BLOCKED) &&
          (\old(state) == READY_FOR_PAYMENT ==>
                state == READY_FOR_PAYMENT ||
                state == READY_FOR_USE) &&
          (\old(state) == BLOCKED ==> state == BLOCKED);
    */

    /*@ invariant state == BLOCKED <==> attempts == 3;
     */
```

Continued in Fig. 3.6

Figure 3.5: Class `CardImpl` with an encoding of the live cycle property (1/2)

Continued from Fig. 3.5

```
    /*@ also
        requires state == FRESH;
        ensures state == READY_FOR_USE && attempts == 0;
     */
    public void initialiseCard() {
    // ..
    //@ set state = READY_FOR_USE;
    //@ set attempts = 0;
    }

    /*@ also
        requires state == READY_FOR_USE && attempts < 3;
        ensures \result ==> state == READY_FOR_PAYMENT && attempts == 0;
        ensures !\result ==> attempts == \old(attempts) + 1;
        ensures !\result && attempts < 3 ==> state == READY_FOR_USE;
        ensures !\result && attempts == 3 ==> state == BLOCKED;
     */
    public boolean enterPin(int pin) {
    boolean result = false;
    // ...
    //@ set attempts = (result? 0 : attempts + 1);
    //@ set state = (result?
    //@                READY_FOR_PAYMENT :
    //@                (attempts == 3 ? BLOCKED : READY_FOR_USE));
    return result;
    }

    /*@ also
        requires state == READY_FOR_PAYMENT;
        ensures state == READY_FOR_USE;
    */
    public void makePayment() {
    // ...
    //@ set state = READY_FOR_USE;
    }

}
```

Figure 3.6: Class `CardImpl` with an encoding of the live cycle property (2/2)

Chapter 4

Abstract Specifications

An important feature of specifications is that they provide abstraction over the concrete implementations. In the previous chapters, we have seen interface `Student` with its (obvious) implementation `CStudent`. However, a more complicated implementation can be imagined using a list of individual results. Figures 4.1, 4.2 contains such an implementation. Notice that it is on purpose that this class does not implement interface `Student`. If we would do that, the class would inherit the specification in terms of the accessor methods, but in this chapter we want to illustrate a different specification technique in terms of abstract state.

This implementation might be correct, but we would typically not want to expose it to the users of the class. Therefore, we define *model variables* that define an abstract state for the class: they are `credits` and `status`. Internally, a different implementation is used, but the specifications of the methods are given in terms of the abstract state – and thus is sufficiently abstract for the user to understand. This idea has originally been introduced under the name *data abstraction* by Hoare [23].

But of course, to make sure that the concrete implementation corresponds to the abstract specification, a link between the two has to be made. For this purpose, the represents clause defines how the value of the abstract variable is defined in terms of the values of the concrete variables. Internally, whenever a specification in terms of abstract variables is encountered, the tools translate this in a specification in terms of the concrete variables, using the represents clause, and then adherence of the implementation w.r.t. this concrete specification is validated.

In fact, the specification approach using model variables would also have been a convenient way to specify the interface `Student`. Because of the flex-

```
class CCStudent {

    public final static int bachelor = 0;
    public final static int master = 1;

    private List<Integer> credit_list;
    /*@ model private int status;
        model private int credits;
    */

    /*@ represents credits <- sum();
        represents status <- (sum() < 180 : bachelor : master);
    */

    /*@ ensures credits == 0;
        ensures status == bachelor;
    */
    CCStudent () {
        credits = new;
        status = bachelor;
    }

    //@ ensures \result == status;
    public int getStatus() {
    if (sum() >= 180) {
        return master;
    } else {
        return bachelor;
    }
    }

    //@ ensures \result == credits;
    /*@ pure */ public int getCredits() {
    return sum();
    }

    public void addCredits(int c) {
    credit_list.add(c);
    }
```

Continued in Fig. 4.2

Figure 4.1: Class `CCStudent` with an alternative implementation for `credits`(1/2)

Continued from Fig. 4.1

```
    private int sum() {
        int sum = 0;
    foreach int i:credit_list {
        sum = sum + credit_list.get(i);
    }
    return sum;
    }
```

```
}
```

Figure 4.2: Class `CCStudent` with an alternative implementation for `credits`(2/2)

ible connection between concrete and abstract state using the `represents` clause, this would not impose any restriction on the internal state of a class implementing the interface.

Sometimes, a represents clause cannot be defined directly as a translation into concrete variables: sometimes just a relation between the abstract and the concrete state can be expressed, or even only a dependency. JML provides an alternative form of represents clauses and also a dependency clause for this. However, using these falls out of the scope of this course.

Model variables are useful in many cases. Typical examples are as in the example above, where a non-standard implementation is used (often for performance reasons), for the specification of interfaces, and when defining classes that encode complex values. A well-known example in the literature is the specification of a class `Decimal` [12]. The class implements decimal variables using a `intPart` and `decPart` variable, but the specification is given in terms of a single model variable that represents the value of the composed decimal number.

To allow to give nice mathematical specifications, JML comes with a library of so-called model classes. These define classes that encode typical mathematical objects, such as sets, functions and bags. Since they are defined as Java (JML) classes, it is possible to use them as type for a model variable, and still relate them to concrete state. Support to reason about

them in mathematically easy way, however, is still a subject of research (see *e.g.*, [14, 20] for some work in this direction).

4.1 On spec_public variables

In Chapter 2 spec˙public variables have been introduced, in order to use private variables in specifications. In fact, declaring a variable `x` as `spec_public` implicitly is equivalent to declaring a model variable, say `_x`, *and* a represents clause `represents _x <- x`. Thus in particular, from the class user's point of view, the two specifications should be equivalent, and the specification only gives an abstract view on the implementation.

4.2 Model versus ghost variables

Finally, it is important to understand the difference between model and ghost variables. Both are variables that are used for specification-purposes only, and they do not occur during the execution of the program.

However, model variables provide an abstract representation of the state. If the underlying state changes, implicitly the model variable also changes. Often it is possible to define this relationship explicitly as a translation, but sometimes it can only be given in a non-constructive manner, or even as a dependency relation.

In contrast, ghost variables extend the state. They provide some additional information that cannot be directly related to the object state. Ghost variables are often used to keep track of the events that have happened on an object, *e.g.*, which methods have been called, how often have these methods been called *etc.*. There also exists work where ghost variables have been used to keep track of the resources used by the program: every time a new object is created, there is an associated set-annotation that increases a resource counter, modelled as a ghost variable [8]. In this way, the specification can state something about the number of objects that have been created by the program. This information allows then to define a resource analysis over the application.

Chapter 5

Static checking of JML annotations

Run-time checking is a useful technique to get quick feedback on whether an application respects its (JML) annotations. However, the drawback of it is that in general it cannot give a 100 % correctness guarantee. For almost any realistic program, it is impossible to get complete coverage by exploring all possible program execution paths. And even when the program's state space is finite, and this might be possible in principle, the performance overhead will be very high.

If we want to have such a high correctness guarantee, we need to use different validation techniques. Program logics have been developed to reason about programs, without actually executing them. The idea of using a program logic and pre- and postconditions to reason about programs dates back to the sixties. Floyd was the first to introduce the concepts of pre- and postconditions, and thinking about program behaviour in an non-executional way. This led Hoare in 1969 to come up with a concrete set of rules to reason about programs [24]. These rules (and any variations thereof) are often called Hoare logic.

5.1 A Quick Overview of Hoare Logic

Given a precondition P, a postcondition Q and a statement S, a *Hoare triple* $\{P\}S\{Q\}$ has the following meaning: if statement S is executed in a state x satisfying precondition P, and if execution of statement S terminates in some state y, then this final state satisfies postcondition Q. Notice the correspondence with the meaning of `requires` and `ensures` clauses in JML:

these are just a more verbose way to write Hoare triples for methods.

The main idea of Hoare's logic is to break down the correctness guarantee of complex statements into correctness guarantees for the components of these statements. Concretely, suppose we compose two statements S_1 and S_2, and we want to show that $\{P\}S_1; S_2\{Q\}$, *i.e.*, if execution starts in a state satisfying P, then after termination of $S_1; S_2$, Q should hold. Then it is sufficient to find an *intermediate predicate* R and to show $\{P\}S_1\{R\}$ and $\{R\}S_2\{Q\}$. Thus correctness of the composed statement is thus reduced to a correctness problem of the components of the composed statements.

As a proof rule, this is expressed as follows:

$$\frac{\{P\}S_1\{R\} \quad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

This should be read as follows: in order to prove the correctness triple below the line, it is sufficient to prove the correctness triples above the line.

Another example of a Hoare logic proof rule is the rule for the conditional statement (`if-then-else`).

$$\frac{\{P \wedge c\}S_1\{Q\} \quad \{P \wedge \neg c\}S_2\{Q\}}{\{P\}\texttt{if } c \texttt{ then } S_1 \texttt{ else } S_2\{Q\}}$$

I.e., if the condition c holds, then provided the precondition P holds, the then-branch S_1 has to guarantee the postcondition Q, if the condition c does not hold, then the else-branch S_2 has to guarantee the postcondition.

A more interesting rule is the assignment axiom. This states that an assignment $x{:=}E$ guarantees postcondition Q, if before the assignment $Q[x\backslash E]$ was true, *i.e.*, Q was any occurrence of x replaced by the expression E was true. This is easiest understood with an example. Suppose that you have an assignment `x := y + 2;`. If afterwards you want that `x > 0`, then this only can be ensured if the value that is assigned to `x` was greater than 0, *i.e.*, the precondition has to be `y + 2 > 0`. Formally, this is expressed by the following rule (where `_[x\E]` denotes substitution of `x` by `E` – formal definition of substitution falls out of the scope of this course).

$$\frac{}{\{Q[x\backslash E]\}x := E\{Q\}}$$

Finally, there are rules for method declarations that state that to prove correctness of a method specification w.r.t. the method implementation, one

has to prove correctness of the method body[1].

$$\frac{\{P\}body\{Q\}}{\{P\}\texttt{void } m\texttt{() } \{body\}\{Q\}}$$

Key feature of Hoare logic is that it allows one to prove something about a method for all possible input states and all possible arguments, and without executing the code. We can prove for example that the body of a method `swap` always swaps the values of the variables `x` and `y`, whatever their initial values. In JML, the specification for this method would be the following:

```
//@ ensures x == \old(y) && y == \old(x);
void swap () {
  int t = x;
  x = y;
  y = t;
}
```

In classical Hoare logic, this would be specified as follows:

$$\{x = A \wedge y = B\}\texttt{swap()}\{x = B \wedge y = A\}$$

A and B are often called logical variables. Implicitly, the correctness triple holds for all possible values of A and B, and as demonstrated by the specification above, their typical use is what is expressed using the `\old` keyword in JML. The correctness proof for this method would then roughly look as follows[2]:

$$\dfrac{\dfrac{\{x = A \wedge y = B\}\texttt{int t = x}\{x = A \wedge y = B \wedge t = A\}}{\{x = A \wedge y = B\}\texttt{int t = x}; x = y\{x = B \wedge y = A \wedge t = A\}} \quad \dfrac{\{x = A \wedge y = B \wedge t = A\}x = y\{x = B \wedge y = B \wedge t = A\}}{\{x = A \wedge y = B \wedge t = A\}x = y; y = t\{x = B \wedge y = A\}}}{\{x = A \wedge y = B\}\texttt{int t = x}; x = y; y = t\{x = B \wedge y = A\}} \quad \dfrac{\{x = B \wedge y = B \wedge t = A\}y = t\{x = B \wedge y = A\}}{\{x = A \wedge y = B\}\texttt{int t = x}; x = y; y = t\{x = B \wedge y = A\}}$$

$$\{x = A \wedge y = B\}\texttt{swap()}\{x = B \wedge y = A\}$$

The Hoare triples as we have seen so far describe *partial correctness relations*: if a method terminates, its postcondition will be established. Sometimes one wishes to specify explicitly that a method *must* terminate. For this purpose, a *total correctness relation* is defined. For this we use the notation $[P]S[Q]$. Such a total correctness triple should be read as follows: if

[1] This rule comes in many variations, for rules with and without return values, parameters etc., but the basic idea is always the same

[2] In fact, a completely formal proof would also require the use of weakening and strenghtening rules. That falls out of the scope of this course; for this we refer to the course on Program Verification.

we execute statement S from a state x in which precondition P holds, *then* execution of statement S will terminate in a state y, and in this state y, postcondition Q holds.

5.2 Mechanising Hoare Logic

Hoare logic as it is is not directly suited for developing tool support. The complicating factor is that one needs to "invent" the intermediate predicates that hold between the composition of two statements. However, in 1976, Dijkstra observed that it was actually not necessary to invent this intermediate predicate; instead one could compute the weakest predicate that would ensure required postcondition. It would then be sufficient to show that the specified precondition implied this weakest precondition. This computation of the weakest precondition is expressed by the rules from the weakest precondition calculus, where $\mathsf{wp}(S, Q)$ denotes the weakest predicate such that $\{\mathsf{wp}(S, Q)\}S\{Q\}$ is a correct triple:

$$\begin{aligned} \mathsf{wp}(S_1; S_2, Q) &= \mathsf{wp}(S_1, \mathsf{wp}(S_2, Q)) \\ \mathsf{wp}(x{:=}E, Q) &= Q[x \backslash E] \\ \mathsf{wp}(\texttt{if } c \texttt{ then } S_1 \texttt{ else } S_2, Q) &= c \Rightarrow \mathsf{wp}(S_1, Q) \wedge \neg c \Rightarrow \mathsf{wp}(S_2, Q) \end{aligned}$$

Thus, instead of "inventing" the intermediate predicate for a statement composition, the weakest precondition calculus "computes" it.

Finally, to show that a method implementation respects its specification, one has to do the following: given precondition P, postcondition Q and method body B, compute $\mathsf{wp}(B, Q)$ and show that $P \Rightarrow \mathsf{wp}(B, Q)$. Notice that both the predicate P and the weakest precondition of the body and the postcondition are predicates in first-order logic. For proving properties in first-order logic, many different automated theorem provers exist.

Thus, by implementing the rules of the weakest precondition calculus, and using an (or multiple) automated first-order theorem prover(s) for the generated proof obligations, an automated program verification tool can be built that allows one to prove that for any possible input state and any possible input parameters, a method respects its specification.

5.3 Automated Program Verification for Java

The ideas of the weakest precondition calculus form the basis for several verification tools for Java, such as Loop [10], Krakatoa [38], Key [9][3], Everest [13], and ESC/Java [17]. Also for other languages similar tools exist, *e.g.*, Code Contracts [37] and Spec# [4].

To apply this approach on a realistic programming language, such as Java, the rules have to be adapted for side effects, exceptions and other sources of abrupt termination, dynamic method binding etc., following the Java semantics, see *e.g.*, [26, 29, 40]. Of course, these tools all focus on sequential programs – to reason about multithreaded programs one needs techniques to specify that one thread's behaviour cannot influence the behaviour of another thread. A possible approach for this is the use of concurrent separation logic, another extension of Hoare logic, that allows to explicitly talk about which parts of the heap are accessed by a code fragment, see *e.g.* [28].

During the System Validation course, we will use the ESC/Java tool, because it is one of the most well-developed tools to reason about Java. In fact, at the end of the nineties, several people were working on verification of Java programs. Java was a nice target language to reason about, because it had a reasonably well-defined semantics, described in the language specification [21]. Initially, the tools that were being developed used different specification languages, but quickly JML emerged as the language of choice for all these tools. Effort has been put in unifying the tools, so that they all use the same specification language (possibly with some tool-specific extensions, preceded by a special keyword).

ESC/Java initially started as a tool that would give fast feedback, by compromising on soundness and completeness. Instead of applying a full weakest precondition calculus, at some points some shortcuts were taken, to be able to give quick feedback, without making large demands on the user. In particular, initially there was no support for reasoning about loops – every loop would simply be approximated by a single iteration in the verifier. However, development of ESC/Java has continued, and now the tool is mostly sound and complete. Some known sources of uncompleteness are the handling of the ranges of primitive types, and a not fully sound and complete semantics of class invariants. However, in practice the results of the tool are very good, and if ESC/Java does not produce warnings on a Java application, then it is highly likely to respect its specification.

[3] In fact, this uses an extension of Hoare logic, called dynamic logic, but the underlying principles are basically the same.

5.4 Reasoning about method calls

An important feature of Hoare logic-like approaches is that the verification is modular. Each method is verified in isolation, and any method call inside a body is approximated by its method specification. As a concrete example, suppose we verify method `activityBonus` from class `CStudent` (in Figure 3.1)[4].

```
/*@ requires 0 <= bonus && bonus <= 5 && active;
    {|requires getStatus() == bachelor;
      ensures getCredits() == (\old(getCredits()) + bonus > 180 ?
                  180 : \old(getCredits()) + bonus);
      also
      requires getStatus() == master;
      ensures getCredits() == \old(getCredits() + bonus);
    |}
*/
public void activityBonus(int bonus) {
    if (active && getCredits() + bonus <= 180) {
      addCredits(bonus);
    } else {
    setCredits(180);
    }
}
```

This implementation calls method `addCredits`. Instead of inlining the implementation of `addCredits`, the verification uses the specification of `addCredits`.

```
/*@ requires c >= 0;
    ensures getCredits() == \old(getCredits()) + c;
*/
public void addCredits(int c);
```

Suppose that we are verifying the case where `getStatus() == bachelor && active && getCredits() + bonus <= 180`. To guarantee the postcondition, one has to show the following:

```
0 <= bonus && bonus <= 5 && active &&
getStatus() == bachelor && \old(getCredits()) + bonus <= 180 ==>
  getCredits() == \old(getCredits()) + bonus ==> (postcondition)
    getCredits() == \old(getCredits()) + bonus
```

[4]Recall that the {|...|} notation is briefly explained in Chapter 3.

In addition, one also has to show that the conjunction of the precondition of `activityBonus` and `activity` implies the precondition of `addBonus` (`bonus >= 0`), which is clearly the case.

However, things are not always that simple. Consider class `Point` in Figure 5.1 and the class `Line` in Figure 5.3. For convenience, only a minimal amount of specifications have been given, as this is sufficient to illustrate the problem.

When we verify method `stretchLine` in class `Line`, we use the specification of method `moveHorizontal`, as explained above. This seems to be sufficient – but it is not! Because the specification of `moveHorizontal` does not state anything about what happens with the value of the `y` field – and thus, we cannot assume anything about `y` after the call to `moveHorizontal`. As we cannot assume anything, we cannot be sure anymore that the line is horizontal, and thus that the `length` method is even defined on it. Thus, the postcondition of `stretchLine` cannot be formerly established.

This problem is known as the frame problem [11, 41]. Basically, the point is that for modular verification one needs to know what is the frame of a method, *i.e.*, what are the variables that may be changed by the method, and what is the anti-frame, *i.e.*, which variables may not be changed by the method.

To specify this, JML has a so-called assignable clause. This provides a list of variable locations that *may* be modified by a method (thus, it may be an over-approximation of the actual set of locations that is modified by the method). By default, any pure method should have an empty assignable clause. An assignable clause can also denote a set of locations; typical examples are `\nothing` (the empty set, thus basically a pure method), `\everything` (any location might be changed), and `a[i..j]`, all elements in the array between indices `i` and `j`. In the example above, we of course should add a clause `assignable x;` to the specification of `moveHorizontal`, thus implicitly saying that `y` may not be changed by this method.

Notice that specifications such as for `getX` and `getY` are really necessary in this case. The specifications in `Line` for the points `begin` and `end` use these methods. The specifications of `getX` and `getY` specify how these methods relate to the variables `x` and `y` in class `Point`. Without this specifications, the verification of `stretchLine` cannot deduce that the outcome of `getY` has not changed[5] Again, it is important to realise that ESC/Java uses only the *specifications* of `getX` and `getY` for verification, it does not look at

[5]Of course, the specifications in class `Line` also could be written in terms of `begin.x`, `begin.y` etc. In that case, the specifications for `getX` would not be necessary.

```
package chapter5;

public class Point {

    private /*@ spec_public */ int x;
    private /*@ spec_public */ int y;

    Point(int x, int y){
        this.x = x;
        this.y = y;
    }

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

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

    void setX(int x) {
        this.x = x;
    }

    void setY(int y){
        this.y = y;
    }

    void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
```

Continued in Fig. 5.2

Figure 5.1: Class `Point`

Continued from Fig. 5.1

```
        assignable x;
        ensures x == \old(x) + dx;
     */
    void moveHorizontal(int dx) {
        x = x + dx;
    }

    /*@ requires dy >= 0;
        ensures y == \old(y) + dy;
     */
    void moveVertical(int dy) {
        y = y + dy;
    }


}
```

Figure 5.2: Class `Point`

their implementation.

Of course, in this particular example, it would be possible to add a postcondition `getY() == \old(getY())`. But in general this is not a satisfactory solution: a class might have many variables and only a few are typically changed by a method. Moreover, when a new variable is added, for every method that does not change it, an additional postcondition about this variable not being changed would have to be added. As one can imagine, this is error-prone, and leads to overly verbose specifications.

Using assignable clauses, we can formulate a weakest precondition rule to verify method calls[6]. Suppose that the call `x.m()` resolves to a method with precondition Pre_m, postcondition $Post_m$ and assignable clause $\mathcal{A}_m$, then roughly the rule looks as follows.

$$\mathsf{wp}(x.m(), Q) = Pre_m \wedge (\forall v \in \mathcal{A}_m . Post_m \Rightarrow Q$$

The easiest way to remember is that every method call to m gives rise to two proof obligations:

[6]Similar rules exist of course for rules with return value, parameters etc.

```
package chapter5;

public class Line {

    /*@ non_null */ Point begin;
    /*@ non_null */ Point end;

    //@ requires b != null && e != null;
    Line(Point b, Point e){
        begin = b;
        end = e;
    }

    //@ ensures \result == (begin.getY() == end.getY());
    /*@ pure */ boolean isHorizontal () {
        return (begin.getY() == end.getY());
    }

    /*@ requires dx >= 0 && isHorizontal();
        ensures isHorizontal();
     */
    void stretchLine (int dx) {
        end.moveHorizontal(dx);
    }
}
```

Figure 5.3: Class `Line`

- The postcondition of the method call has to ensure the postcondition Q, where for all variables in the assignable clause of the method, nothing is known about their values, except what is specified in the postcondition of m.
- The weakest precondition of the code preceding the method call w.r.t. the precondition of m has to be implied by the precondition P.

5.5 Statement Annotations - Helping the Verifier

Sometimes, the program verifier has to get some guidance. In exceptional cases, a complex intermediate predicate has to be given explicitly, using an `@assert` P`;` annotation. This is for example necessary when complex calculations are made, and the automated theorem provers need some guidance on how to reason about them. Every `@assert` P annotation gives rise to two proof obligations:

- the precondition of the method that is being verified has to imply the weakest precondition of the code preceding the assertion and postcondition P;
- P has to imply the weakest precondition of the code following the assertion and the postcondition.

For straightline code, such annotations are almost never necessary. One exception in the literature is the verification of addition and multiplication of class `Decimal` [12].

However, for code that contains loops, such annotations are almost always necessary. To explain this, we first present the Hoare logic rule for while-loops.

$$\frac{\{c \wedge I\} S \{I\}}{\{I\}\texttt{while } c \texttt{ do } \quad S\{\neg C \wedge I\}}$$

This rule features a so-called *loop invariant* I: a predicate that is preserved by every iteration of the loop. To show that the invariant is preserved by every iteration of the loop, one has to show that always if the condition holds – and thus the loop body will be executed once more – if the invariant holds before the loop body is executed, then it will also hold after the loop body has terminated. From this we can conclude that if the whole loop is executed in a state in which the loop invariant holds, then after termination

of the loop, the loop invariant still holds, and in addition the negation of the loop condition holds[7].

In general, such a loop invariant cannot be found automatically. Instead the user is supposed to specify it (nevertheless, for simple loops ESC/Java can make some good guesses). Specifying a loop invariant basically gives rise to three proof obligations:

- the precondition of the method has to be sufficient to guarantee that when the loop starts, the loop invariant holds, *e.g.*, if the loop is preceded by a statement S_1, then $P \Rightarrow \mathsf{wp}(S_1, I)$ has to hold;
- the loop body has to preserve the loop invariant, thus $c \wedge I \Rightarrow \mathsf{wp}(S, I)$;
- $\neg c \wedge I$ have to imply the weakest precondition of the code after the loop body and the postcondition of the method, *i.e.*, if the loop is followed by a statement S_2, then $\neg c \wedge I \Rightarrow \mathsf{wp}(S_2, Q)$ has to hold.

Figure 5.4 shows two examples of non-trivial loop invariants. The first method computes $\mathtt{n}^3$ without actually using the power function. Its loop invariant describes the intermediate values for all local variables. The second method checks whether a given value occurs in an array. The loop invariant expresses the property that holds for all elements in the array that have been examined so far.

5.6 Termination

Finally, static checking can also be used to prove termination of methods. One could consider this as a simple liveness property, and thus this is completely out of reach for a run-time assertion checker. In Section 2.2 we have discussed that methods could be specified with a `normal_behavior` keyword, implicitly saying that the method would have to terminate.

For straightline code without loops, proving termination is obvious. However, for code with loops, again the user has to provide some information. Above we saw that a variant of Hoare logic exists, defining total correctness. For most statements, the rules for partial correctness and total correctness are identical. However, for loops the total correctness rule is different. In addition to the invariant I, it also uses a variant expression V. This variant expression has to be well-founded, *i.e.*, it cannot decrease forever. For every

[7] In fact, to reason about Java, variations of this rule exist, allowing to reason about loops that terminate abruptly *e.g.*, because of an exception or a `return` statement.

iteration of the loop, one has to show that the variant is strictly decreasing (for this end, in the rule a logical variable v is used to remember the old value of the variant). Since the variant is a well-founded expression, it cannot decrease forever, and thus the loop has to terminate.

$$\frac{[c \wedge I \wedge V = v]S[I \wedge V < v]}{[I]\texttt{while } c \texttt{ do } S[\neg c \wedge I]}$$

As for the invariant, the loop variant in general cannot be found automatically, it has to be specified explicitly by the annotater/developer (although there are also several tools available that can suggest variants).

Consider again the loops in Figure 5.4. If we wish to show in addition that the loop terminates, we have to add a variant to it, using the JML `decreasing` keyword. For `third_power` an appropriate choice would be: `decreasing n - k;`, for `search`, `decreasing a.length - i;` would work.

A similar approach as for loops is also used to reason about correctness and termination of recursive calls.

```
package chapter5;

public class LoopExamples {

    /*@ requires n >= 0;
        ensures \result == n * n * n;
     */
    public int third_power(int n) {
        int u = 0;
        int v= 1;
        int w = 6;
        int k = 0;
        //@ loop_invariant 0 <= k && k <= n;
        //@ loop_invariant u == k * k * k;
        //@ loop_invariant v == 3 * k * k + 3 * k + 1;
        //@ loop_invariant w == 6 * (k + 1);
        while (k < n) {
            u = u + v;
            v = v + w;
            w = w + 6;
            k = k + 1;
        }
        return u;
    }


    /*@ requires a != null;
        ensures \result ==
                 (\exists int i; 0 <=i && i < a.length; a[i] == val);
     */
     public boolean search(int [] a, int val) {
        boolean found = false;
        int i = 0;
        /*@ loop_invariant
             found == (\exists int j; 0 <= j && j < i; a[j] == val);
            loop_invariant 0 <= i && i <= a.length;
            loop_invariant a != null;
        */
        while (i < a.length & !found) {
            if (a[i] == val) {found = true;}
            i++;
        }
        return found;
    }

}
```

Figure 5.4: Loops with non-trivial loop invariants

Bibliography

[1] S. M. K. O. Abadi, C. Bockisch, and M. Ak sit. Applying the composition filter model for runtime verification of multiple-language software. In *The 20th annual International Symposium on Software Reliability Engineering,ISSRE 2009*, pages 31–40. ieee, 2009.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [7], pages 151–171.

[5] M. Barnett, D. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In E. Poll, editor, *Workshop on Formal Techniques for Java Programs*, pages 11–19, 2004.

[6] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In K. Havelund and G. R. su, editors, *ENTCS*, volume 55(2). Elsevier Publishing, 2001.

[7] G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[8] G. Barthe, M. Pavlova, and G. Schneider. Precise analysis of memory consumption using program logics. In *Software Engineering and Formal Methods*, pages 86–95. IEEE Press, 2005.

[9] B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[10] J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[11] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, 1995.

[12] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 55:53–80, 2005.

[13] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In D. M. K. Araki, S. Gnesi, editor, *Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.

[14] J. Charles. Adding native specifications to JML. In *Workshop on Formal Techniques for Java Programs*, 2006.

[15] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, 2003.

[16] Code contracts. http://research.microsoft.com/en-us/projects/contracts/.

[17] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In Barthe et al. [7], pages 108–128.

[18] Á. Darvas and R. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

[19] A. Darvas and P. Müller. Reasoning about method calls in JML specifications. In *Workshop on Formal Techniques for Java Programs*, 2005.

[20] A. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, 2(6):477–499, Dec. 2008.

[21] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, third edition*. The Java Series. Addison-Wesley Publishing Company, 2005.

[22] K. Havelund and G. Rosu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24, 2004.

[23] C. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[25] E. Hubbers and M. Oostdijk. Generating JML specifications from UML state diagrams. In *Forum on specification & Design Languages*, pages 263–273. University of Frankfurt, 2003. Proceedings appeared as CD-Rom with ISSN 1636-9874.

[26] M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. PhD thesis, Computing Science Institute, University of Nijmegen, 2001.

[27] M. Huisman and A. Tamalet. A formal connection between security automata and JML annotations. In *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 340–354. Springer-Verlag, 2009.

[28] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice Sophia Antipolis, 2009.

[29] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58(1–2):61–88, 2003.

[30] The JASS project. `http://semantik.informatik.uni-oldenburg.de/~*jass/`.

[31] G. Leavens, A. Baker, and C. Ruby. Preliminary Design of JML: a Behavioral Interface Specification Language for Java. Technical Report 98-06, Iowa State University, Department of Computer Science, 1998. `http://www.cs.iastate.edu/~*leavens/JML/prelimdesign/`.

[32] G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual. Preliminary draft `http://www.cs.iastate.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`.

[33] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[34] K. Leino, G. Nelson, and J. Saxe. ESC/Java user's manual. Technical Report SRC 2000-002, Compaq System Research Center, 2000.

[35] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004. Available from `www.sct.inf.ethz.ch/publications/index.html`.

[36] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[37] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In L. Hendren, editor, *International Conference on Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2008.

[38] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program certification, 2003. Manuscript.

[39] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd revised edition, 1997.

[40] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, pages 131–140. Fernuniversität Hagen, 1999. Technical Report 263, Available from `sct.inf.ethz.ch/publications`.

[41] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. Technical Report 01-03, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, April 2001.

[42] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[43] J. Niño and F. Hosch. *An introduction to Programming and Object-Oriented Design Using Java*$^{\mathrm{TM}}$. Wiley, 2008. Third edition.

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

[45] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005.

[46] H. Rebêlo, R. Lima, G. Leavens, M. Cornélio, A. Mota, and C. Oliveira. Optimizing generated aspect-oriented assertion checking code for jml using programming laws: An empirical study, 2010. Submitted.

[47] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3:30–50, 2000.

[48] D. Zimmerman and R. Nagmoti. JMLUnit: The Next Generation. In B. Beckert and C. Marché, editors, *1st International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, volume 6528 of *Lecture Notes in Computer Science*. Springer, 2010.