Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Theoretische Informatik

Diplomarbeit

Formal Semantics for the Java Modeling Language

Daniel Bruns

8. Juni 2009

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Mattias Ulbrich, Benjamin Weiß

Danksagung

Mein Dank gilt Alexander Butsch, der diese Arbeit Korrektur gelesen hat, sowie Erik Burger und Malte Cornils, die zumindest den Willen dazu, aber dann leider nicht die Zeit dazu hatten. Dank auch an Gary Leavens für die schnellen Antworten zu meinen Fragen und an Tina Grahl für die permanente Inspiration und die Debatten über Alles, was *nicht* mit JML zu tun hat. Letztendlich geht mein besonderer Dank an meine Betreuer, Prof. Peter Schmitt, Mattias Ulbrich und Benjamin Weiß, mit denen ich über mehrere Stunden angeregte Debatten führen konnte, durch die ich immer wieder in meiner Arbeit bestätigt fühlen konnte.

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Daniel Bruns
Karlsruhe, den 8. Juni 2009

Deutsche Zusammenfassung

Diese Arbeit beschäftigt sich mit der Semantik der Java Modeling Language (JML). JML is eine weitverbreitete Spezifikationssprache, die speziell auf Java zugeschnitten ist und sowohl zur statischen als auch zur Laufzeitanalyse von Programmen verwandt wird. Bislang beschränkt sich die offizielle Spezifikation von JML auf eine weitgehend verbale Definition der Semantik, die zudem teilweise unvollständig oder widersprüchlich ist. Daraus erwächst unter anderem, dass die verschiedenen Software-Werkzeuge, die JML implementieren, Sprachelemente der JML unterschiedlich auslegen.

Bisherige Arbeiten zur formalen Semantik basieren auf Logiken höherer Ordnung oder dynamischen Logiken. In dieser Arbeit wird ein Ansatz vorgestellt, der nur auf elementaren mathematischen Konzepten wie Mengen, Funktionen und Prädikatenlogik erster Ordnung beruht. In dieser Arbeit wird eine simple Abstraktion einer Maschine vorgestellt, auf der der semantische Unterbau basiert, auf welchen Ausdrücke und Spezifikationen von JML abgebildet werden, um eine nahezu vollständige Übersicht der Spezifikationen für sequentielle Java-Programme zu entwickeln.

Contents

Chapter 1

Introduction

This thesis is concerned with the Java Modeling Language (JML), a widespread specification language for Java, which is used in both static and runtime analysis of programs. Yet, the official reference mostly lacks semantics, while several tools which implement JML do not agree on their interpretation of JML specifications. Past approaches have all centered around a certain verification tool and are highly depending on specific higher order logics. In this work, we develop a formalization with little requirements. Upon a simple machine model we describe JML artifacts in basic notations such as first order logic. In that, we provide a nearly comprehensive overview of features which cover nearly all specification elements for sequential Java programs.

1.1 The Java Modeling Language

The *Java Modeling Language* [LBR99, LBR03, LC05, LCC$^+$05, CKLP06], or JML for short, is a formal specification language especially tailored to the Java programming language [AG96]. JML is targeted to provide a comprehensive specification of both *interfaces* (syntax) and *behavior* (semantics)[1] for every aspect of Java and, at the same time, to retain an easy-to-read format. [LBR03] outlines these three major goals:

> - "JML must be able to document the interfaces and behavior of existing software, regardless of the analyses and design methods to create it. [. . .]
> - The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training. [. . .]

[1]Therefore, some authors use the term "Behavioral Interface Specification Language" (BISL). [CL94, HLL$^+$09]

- The language must be capable of being given a rigorous formal semantics, and must also be amendable to tool support."

This enumeration shows that one aim is, to have a language which is (syntactically) close to Java, not too abstract like specification languages like Z [ASM80]. It is also primarily intended to specify existing code, rather than to implement programs according to a preexisting specification.

JML Specifications are written directly into the source files of Java modules. The basic syntax of expressions in JML is – with some extensions – the same as in Java. By this, it is expected from the targeted user (programmers, not theoretically trained) to develop an intuitive understanding of the semantics of the language. And, as specification are always given inside comments (similar to Javadoc), JML constitutes as a proper extension to the Java language which does not interfere with the execution of the program.

Unlike the *Object Constraint Language* (OCL) [OCL05], which is part of the UML standard, JML is not regulated by some governing body, but rather constitutes as a community effort led by Gary T. Leavens at the University of Central Florida.

Uses of JML

As stated above, JML primarily serves the purpose of documenting artifacts of the Java language. There is nothing like *the* application for JML, but rather a few. This includes both static type checking, run-time assertion checking as well as formal verification methods [LCC+05]. There are also other tools which produce JML specifications as output, like the *Daikon* tool for loop invariant detection. For an overview of JML tools, refer to [BCC+05]. Some of them are discussed in Sect. 5.1.

Since Java is by far one of the most popular programming languages, being "simple, object oriented, and familiar" [GM95], JML gained significant popularity over the past few years, too. In fact, it is much more accepted among programmers than Z, OCL etc. On many parts, this fact might rely on the syntactical similarities between both languages.

The claimed primary antecedent of JML is the *Larch* specification language [GHW85, Lea99] which follows a similar approach for the C++ programming language. In addition, several features are taken from the *refinement calculus* [BvW98]. The approach of writing specifications in the specified language itself (or a similar looking one), is in part inspired by languages like *Eiffel* [Mey92b].

Language structure

JML can be subdivided into a common expression language,[2] which is very similar to Java, and on top of this, several *specification elements* which are meant to describe the behavior of the program in different ways. This includes "classical" features such as *assertion statements* within the program code (for run-time checking). As more "high-level" elements, there are *method specifications* based on the *design-by-contract* paradigm [Mey92a]. In few words, this means that a method call is regulated by a contract: A caller guarantees some preconditions and the receiver in turn guarantees some conditions to hold after the execution of the method. Finally, there are *type specifications* such as invariants which have to be preserved throughout the run of the program. JML also features variables and fields which may be additionally used in specifications (model fields, ghost variables/fields).

1.2 JML by example

In this section, we will take a first look on an example specification in JML. This only features basic syntax elements and is intended to create some intuitive understanding. We share no interest in implementation details – those are omitted – but in according specifications.

Consider a small library with some books for lend. In our first considerations, a book can be lent or not and the library gives it away as long as it is present. To be present in this context means, the library owns it and it is not lent at the time. This leads to the first specification shown in Fig. 1.1.

This is basically Java code with a most basic method specification for `lend`. This method specification is directly attached to the method declaration in Line 13 and is enclosed in a block comment. Only comments beginning with an at-symbol (`@`) are taken to be JML specifications. A leading at-symbol in every line is not required by syntax, but used by convention. Every specification clause has to be terminated with a semicolon (`;`), like declarations in Java.

The method specification begins in Line 8 with a Java-style privacy modifier and the keyword `normal_behavior`. The latter requires the method to terminate normally, i.e. without exceptions. The precondition, i.e. the caller's obligation to establish, is marked with `requires` and the postcondition, i.e. the receiver's guarantee in the post-state, with `ensures`. The appearing logical operators `!` (negation) and `&&` (short-circuit conjunction)

[2]We denote the full expression language by *JML-*E_1, and a sub-language which can be evaluated in one state by *JML-*E_0.

```
1     public class Book {
2         private /*@ spec_public @*/ boolean lent;
3     }

5     public class Library {
6         private Collection coll;

8         /*@ public normal_behavior
9           @      requires coll.contains(b)
10          @               && !b.lent;
11          @      ensures  b.lent;
12          @*/
13        public void lend (Book b);
14    }
```

Figure 1.1: A small library which lends books at good-will.

are used as in Java. It is also allowed to use side-effect free method invocations as shown in Line 9. Thus, the meaning of this method specification is: If the book is available, then any invocation of method `lend` terminates successfully with the book being lent then. Privacy modifiers are not inside the center of our concern, JML syntax rules however require specifications to be at least as private as the fields they address. To use a public specification, we are forced to assign to the given fields specification-only privacy modifiers like `spec_public`.

To make things more interesting, we modify this example in that `lend` returns a numbered receipt represented by an integral value. The library keeps a global (i.e., `static` in Java/JML terms) counter `rctCnt` on receipt numbers, which is shared between all libraries in the universe, and is increased every time a book is successfully lent to avoid duplicate numbers. To implement these amendments, we use special JML expressions to add two postconditions:

```
//@ ensures \result == rctCnt;
//@ ensures rcpCnt == \old(rcpCnt) +1;
```

Both expressions are prefixed with a backslash (\) to distinguish them from other (Java-compatible) expressions. `\result` as the name suggests, stands for the return value of the specified method. The `\old` expression is applied on an arbitrary expression to yield the value of the pre-state. In this case, it means that `rctCnt` is incremented by 1 during the execution of `lend`. Of course, both expressions only make sense in a context in which there exists a previous state to refer to. Therefore, they may not be used in the precondition. At last to notice, both lines are given within Java end-of-line comments, again beginning with an at-symbol.

So far, we have only specified the behavior of `lend` in a successful case. But what if the (likewise common) case applies, in which a customer requests a book which is already lent? In this case, the book cannot be lent again and the counter is not stepped. Furthermore, it has to be signaled to the customer that this was not possible. The Java way – and thus the JML way – to achieve this, is to throw an exception. Thus, the lines shown in Fig. 1.2 are added to our method specification.

```
    /*@ public exceptional_behavior
      @      requires !coll.contains(b)
      @                || b.lent;
      @      signals (BookNotFoundException e)
      @                b.lent == \old(b.lent) &&
      @                rctCnt == \old(rctCnt);
      @*/
```

Figure 1.2: An exceptional case added.

This specification case looks much like the one discussed above with the exception that it begins with `exceptional_behavior` and the postcondition is marked with `signals`. The `signals` clause describes the post-state *in case* the method invocation was unsuccessful. This is a sufficient condition, not a necessary one, i.e. it does *not require* the method to throw an exception.

This however, is implicitly included in the declaration `exceptional-_behavior` which excludes normal behavior. Likewise, the `normal_behavior` declaration above is given to exclude exceptional behavior. Both of them require the method to terminate. These declarations can be "desugared" to a more general form which we will be using in Sect. 4.2.1.

Finally, we will make some use of class specifications. We could think of some invariant which postulates that all costumers of our library are so very thoughtful, that no book is ever lost. As it turns out, this property cannot be represented as an invariant. Not because, we could not imagine such a library, but because "not lost" expresses some temporal property. As invariants are meant to represent a "time-less fact", they are not the specification element of choice. Happily, JML also features *history constraints* which are meant to be used in those situations to relate two points in time with each other. Let us add the following line to the class declaration of `Library`:

```
/*@ public constraint (\forall Book x;
  @         \old(coll.contains(x)); coll.contains(x));
  @*/
```

In this example we make use of the `\old` expression again to compare two points in time somewhere during the existence of our library. As it can be seen, it may be applied to a more complex expression than just to a reference. The newly introduced construct in this line is the quantifier `\forall`. Its meaning is quite straight-forward: Every book which was contained in the collection before, is contained in the collection right now. The two expressions following the binding of the variable can be read as a guard condition.

We can still add some invariant, say, to postulate that the collection contains at least two different books. Using existential quantification, this looks like that:

```
/*@ public invariant (\exists Book y, z; y != z;
  @           coll.contains(y) && coll.contains(z));
  @*/
```

The final version of our example can be seen in Fig. 1.3. We have conjoined the multiple postconditions and put the two specification cases together with the keyword `also`.

1.3 Goal and approach of this work

As it was said in the introductory section, JML is meant to describe the semantics of Java artifacts in a rigorous way. This raises – most naturally – the question for evenly rigorous, formal semantics of JML in turn. (In fact, this is backed up by the third major goal of page 2.) Most of JML is specified in its reference manual [LPC$^+$08]. Although it is quite voluminous already, it is at the time of writing still in a draft version. While formal syntax definitions are comprehensive, at several occasions, there are still some non-trivial holes in semantical explanations. If semantics are given, their quality usually ranges between informal intentions and a verbal explanation of formal properties at best. These are often unclear or ambiguous. In addition, there are several parts in which semantics are disputed even among the community.

On the other hand, there are several tools which implement JML, or rather a subset of it. But even on a common language subset, these tools do often disagree on semantics. Furthermore, interpretations are depending on the target language. Most approaches use complex higher order logics, which are not easily interchangeable (or their translations). This emerges the need for semantics which are both rigorously formal and at the same time independent from other formal languages.

```
public class Library {
    private Collection coll;
    private static int rctCnt;

    /*@ public constraint (\forall Book x;
      @     \old(coll.contains(x)); coll.contains(x));
      @ public invariant (\exists Book y, z; y != z;
      @     coll.contains(y) && coll.contains(z));
      @*/


    /*@ public normal_behavior
      @     requires coll.contains(b)
      @             && !b.lent;
      @     ensures  b.lent && \result == rctCnt
      @       && ensures rcpCnt == \old(rcpCnt) +1;
      @ also
      @ public exceptional_behavior
      @     requires !coll.contains(b)
      @             || b.lent;
      @     signals (BookNotFoundException e)
      @             b.lent == \old(b.lent) &&
      @             rctCnt == \old(rctCnt);
      @*/
    public int lend (Book b);
}
```

Figure 1.3: The same library with at least two different books and very well-behaved customers, who never lose a single book.

In this work we will only rely on elementary mathematical notations such as sets, functions and first order predicate logic. By this, the given formal semantics are widely understandable without any further introduction to logical calculi or internal details of some implementing tool. At the same time, we try to keep the part in which Java is involved, and in particular how it is processed, rather small. Therefore, we propose a simple machine model which serves as the semantical foundation of JML expressions.

Predominantly, the semantics given throughout this work are based on the semi-formal descriptions found in the JML Reference Manual. On some occasions, these are supplemented by teleological reasoning founded on the available white papers, such as [LBR03], or replaced by our own considerations which are mostly based on analogies to the semantics of Java.

1.4 Outline

The subsequent chapters are organized as follows:

- Chapter 2 contains the foundations for an elaboration on JML. In particular, we develop a formal notion of a system state and describe the interaction between Java and JML. This is done using a black box (nevertheless strong) interface to a virtual machine. This chapter is meant to encapsulate every aspect of the Java language so that it may be referred to it in later parts.

- Chapter 3 describes the sub-language of expressions. At first, we present different evaluation functions and several approaches for a definition of validity. In the following few sections, we discuss the implementation of more complex features like model fields, axioms and data groups. In the remainder of that chapter, we discuss the evaluation of several expressions in JML aligned with several examples. A comprehensive enumeration of expressions can be found in Appendix A.

- Chapter 4 elaborates on the various specification elements. It is structured from "more specific" to more general semantics. It starts with specifications specific to classes or interfaces, namely invariants and history constraints. Then, it covers method specifications with pre- and postconditions and frame conditions. It concludes with specifications which are directly annotated to the runnable part of the program code, i.e. assertions and loop invariants and variants. All of this builds upon the definition of the black box (Chapter 2) and the definition of validity from Chapter 3.

- Chapter 5 begins with a treatise on related work, then summarizes our results and concludes with directions for further work.

Chapter 2

Semantic Foundations for Java Programs

As explained in the introduction, this thesis puts its focus on the Java Modeling Language, not on the Java programming language. Of course, JML can be seen in some sense as an extension to Java, or at least that it relies on Java in various contexts. Therefore, Java cannot be blanked out completely. The goal of this chapter is to cover every issue concerning Java, so that we can refer to it later. On the other hand, it contains very little JML-only features; this mainly includes the types which are added by JML to the otherwise common type structure.

The principle concept is, to regard the execution of Java code as a black box. In particular, throughout the later chapters, we will never talk about certain passages of code, not to mention single assignments or operations of the underlying machine. We are not interested in how the Java code is processed *internally*, but rather which *behavior* is *observable*. To concretize this, we impose strong assumptions on our black box in order to obtain sufficient information on the functional behavior of the program.

To start from, we have still to cover the basic definitions such as a program, identifiers or types. Then we develop a simple notion of a system state based on an abstraction of Java's memory model.

2.1 Java programs

Under a *Java program* Π we understand a fixed collection of compilation units. It is assumed to be syntactically correct according to Language Specification [GJSB00], i.e. compile correct. We only consider *closed programs*, which do contain all necessary information by themselves. In the pursuit of this work, we will fixate a certain program Π.

A program consists of a set of *classes* and *interfaces* $\mathcal{C}$. This includes both classes and interfaces defined in the program itself as well as imported ones, e.g. from `java.lang`. For reasons of simplicity, classes and interfaces are identified with their identifiers, e.g. we say "`Object` is a class". The program is structured in *packages*. The set of packages $\mathcal{P}$ is a partition of $\mathcal{C}$.

A *code fragment* π is a statement block from the source code of a class $C \in \mathcal{C}$. (This implies both syntactical completeness and executability.) It will typically be the body of a method. A code fragment π may be annotated by a set A of JML in-code annotations (see Sect. 4.3). π^A is called the *annotated* code fragment.

The set of valid identifiers $\mathbb{I}$ is given as the union of the set of natural numbers[3] (for array access) and a non-empty, countably infinite set of alphanumerical strings as defined in [GJSB00, Sect. 3.8] (for object and primitive type references).

We say $x \in \mathbb{I}$ is a field (of type T) in class (or interface) C if it is declared through $T\ x$ in the corresponding portion of the source code. If we want to refer to the type of a field definition x, we will write $\mathit{typeof}(x)$. Analoguously, a method with identifier $m \in \mathbb{I}$ is declared in a class.

2.2 Types and values

Types

JML types include both Java types (primitive, array and reference types) as well as types `\bigint` and `\real` representing the mathematical integers and real numbers respectively. There is also a special type `\TYPE` (in capital letters) to represent the collection of Java types. The JML reference manual describes it as being "equivalent to `java.lang.Class`".

Definition 2.1 (Type).

- `boolean` is a type.
- `long`, `int`, `short`, `byte`, `char` are types called *integral types.*
- `double`, `float` are types called *floating point types.*
- `\bigint`, `\real` are types.

[3]The set of natural numbers $\mathbb{N}$ always contains 0. This coincides with the definition of arrays in Java, where the first element is addressed as 0. If 0 should be excluded, we will write $\mathbb{N}_+$.

- If C is a class or interface in Π, then C is a type called *reference type*.
- `\TYPE` is a type.
- If T is a type, then T`[]` is also a type called *array type*.

Integral and floating point types along with `\bigint` and `\real` are also called *numerical types*. Numerical types and boolean are called *primitive types*. Integral, floating point and reference types along with the array types constructed from them are *Java types*. The set of all types defined above is denoted by $\mathcal{T}$. Note that it is also allowed to construct array types from `\TYPE`.

The reflexive and transitive subtype relation is denoted by $\sqsubseteq$. It is the exact same as in Java [GJSB00, Sect. 8.1] and defined for the program by declarations of class inheritance and interface implementation. This definition is meant to be complete, e.g. for a numerical type N and a reference type R it is both $N \not\sqsubseteq R$ and $R \not\sqsubseteq N$. Even so, `\bigint` and `\real` are not super-types of any numerical type. Numerical types may however be converted to each other. `\TYPE` and the array types derived from it are incomparable with any other type. As in Java, interfaces and array types (except for `\bigint[]`, `\TYPE[]`, etc.) are subtypes of `Object`. An excerpt of JML type hierarchy can be seen in Fig. 2.1.

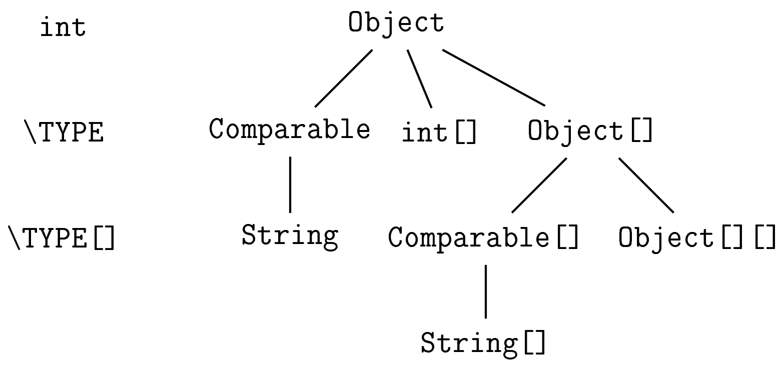

Figure 2.1: JML type hierarchy. Genuine JML types are unrelated to Java types.

Values

For every primitive type T of the Java language we define a value set or *domain* V_T according to [GJSB00]. These are displayed in Tab. 2.2 along with the JML-defined primitive types. Note, that we postulated all integral types to share a common domain, namely a subset of mathematical integers. Likewise, (numerical) floating-point type values are contained in the set of rational numbers.

For reference types however, we need to make a distinction between a *direct* instance of a class and the set of domain elements which are *applicable* to the class/interface definition. This distinction reflects the distinction between *dynamic* and *static* types in Java. The domain of `\TYPE` contains all *Java types*, including null and void, which are not types according to our definition, but included in `java.lang.Class`.[4]

Definition 2.2 (`\TYPE` values).

$$\begin{aligned} V_{\texttt{\textbackslash TYPE}} := &\{T \in \mathcal{T} \mid T \sqsubseteq \texttt{Object}\} \cup \\ &\{\texttt{boolean}, \texttt{long}, \texttt{int}, \texttt{short}, \texttt{byte}, \texttt{char}, \texttt{double}, \texttt{float}, \texttt{null}, \texttt{void}\}\end{aligned}$$

For every primitive type P, it is $V_P^0 := V_P$.

$$\begin{array}{lcll}
V_{\texttt{boolean}} & := & \{true, false\} & \\
V_{\texttt{\textbackslash bigint}} & := & \mathbb{Z} & \\
V_{\texttt{long}} & := & \{-2^{63}, \ldots, 2^{63}-1\} & \subset \mathbb{Z} \\
V_{\texttt{int}} & := & \{-2^{31}, \ldots, 2^{31}-1\} & \subset V_{\texttt{long}} \\
V_{\texttt{short}} & := & \{-2^{15}, \ldots, 2^{15}-1\} & \subset V_{\texttt{int}} \\
V_{\texttt{byte}} & := & \{-2^{7}, \ldots, 2^{7}-1\} & \subset V_{\texttt{short}} \\
V_{\texttt{char}} & := & \{\texttt{u0000}, \ldots, \texttt{uffff}\} & \\
V_{\texttt{\textbackslash real}} & := & \mathbb{R} & \\
V_{\texttt{double}} & := & \mathbb{Q}_{\texttt{double}} \cup \{-\infty, +\infty, NaN\} & \\
V_{\texttt{float}} & := & \mathbb{Q}_{\texttt{float}} \cup \{-\infty, +\infty, NaN\} & \subset V_{\texttt{double}}
\end{array}$$

where $\mathbb{Q}_{\texttt{double}} = \{\pm m \cdot 2^e \mid m \in [0, 2^{53}-1], e \in [-1074, 971]\} \subset \mathbb{Q}$
and $\mathbb{Q}_{\texttt{float}} = \{\pm m \cdot 2^e \mid m \in [0, 2^{24}-1], e \in [-149, 104]\} \subset \mathbb{Q}_{\texttt{double}}$

Table 2.2: Values of JML primitive types.

Definition 2.3 (Reference and array values).

1. For every non-abstract class C the domain is a countably infinite set $V_C^0 := \{c_1, c_2, \ldots\}$.

2. For every abstract class or interface A the domain is empty: $V_A^0 := \emptyset$.

[4]The reference manual is not clear on whether to include them. In fact, the "type" `void` has no use after all. But we tried to keep up the analogy to `Class`, as anticipated.

3. For every array type T`[]` the domain is a countably infinite set $V^0_{T\texttt{[]}} := \{a_1, a_2, \ldots\}$.

4. Domains are mutually exclusive: For types $T_1 \neq T_2$ it is $V^0_{T_1} \cap V^0_{T_2} = \emptyset$.

5. For every class, interface or array type T we define a set of compatible domain elements
$$V_T := \bigcup_{D \sqsubseteq T} V^0_D \mathbin{\dot\cup} \{\text{null}\}$$

The null element is included in any V_T, but excluded from every V^0_T. If we want to exclude null from the set of compatible elements, we will write $V^-_T := V_T \setminus \{\text{null}\}$ for short. The domain sets for (non-abstract) classes and array types need to be infinite due to the arbitrary number of instances which may eventually be created. (See Sect. 2.4.1 for more on object creation.) A domain element of reference type will occasionally be identified with an (semantical) *object*.

The (disjoint) union of all domain sets defined above and additionally including null is called the *universe* $\mathcal{U}$ of the program Π:

$$\mathcal{U} := \bigcup_{T \in \mathcal{T}} V^0_T \cup \{\text{null}\}$$

2.3 System states

We will now model a system state through a functional abstraction of Java's two memory structures – Heap and Stack – as well as a call stack. We take an abstract view of a memory as a mapping of locations to domain elements. This approach was presented in [Ulb08].

Heap

Definition 2.4 (Location)**.** A *location* is a tuple $(o, x) \in \mathcal{U} \times (\mathbb{I} \cup \{\texttt{\textbackslash created}\})$ of a domain element of reference or array type and an identifier representing a field.

1. To a reference type T and a (possibly static, possibly ghost) field x in T there is a location (o, x) for every $o \in V_T$.

2. To an array type T`[]` there is a location (o, n) for every $o \in V_{T\texttt{[]}}$ and $n \in \mathbb{N}$. There is also a location $(o, \texttt{length})$ referencing the length of the array.

3. $(o, \texttt{\textbackslash created})$ is a location for every $o \in V_{\texttt{Object}} \cup V_{T[]}$ for some type T, indicating whether it has been created.

With this definition, a type T inherits all fields (including static ones) from its super-types. Since `\bigint[]`, `\TYPE[]`, etc. are *not* sub-types of `Object`, they do not inherit anything. Nevertheless, the fields `length` and `\created` also exist for those non-Java array types.

The least set containing all locations according to this definition will be denoted by $\mathcal{L}$. This definition explicitly includes the null element. In this way it is guaranteed for any type to have at least one applicable domain element and there is at least one location for a static field. To achieve a most general notation, we include both not yet created to the locations of concern, but their values are underspecified[5].

Locations which are persistent throughout program execution reside on the Heap. The Heap can be seen as a representation of a global system state while no execution is in progress.

Definition 2.5 (Java Heap)**.** A *Heap description* is a function $h : \mathcal{L} \to \mathcal{U}$ satisfying the following constraints:

1. $h(o, x) \in V_{typeof(x)}$ and $h(o, \texttt{\textbackslash created}) \in \{true, false\}$ for $o \in V_{\texttt{Object}}$.
2. $h(a, n) \in V_T$ and $h(a, \texttt{length}) \in V_{\text{int}} \cap \mathbb{N}$ for $a \in V_{T[]}$.
3. If s is a static field in type C, then $h(o_1, s) = h(o_2, s)$ for any two $o_1, o_2 \in V_C$.

Stack

Not all values are stored on the heap. *Variables* in Java are part of the state of a running execution and stored on the stack. These include local variables and method parameters. In any case, they are local to the current method frame. For any variable which does not belong to the current method frame, its value is underspecified.

Definition 2.6 (Java Stack)**.**
A *Stack description* is a partial function $\sigma : \mathbb{I} \nrightarrow \mathcal{U}$.

Definition 2.7 (Update)**.** Let $\theta : \mathbb{I} \nrightarrow \mathcal{U}$ be a partial function, then the following is also a stack description:

$$\sigma^{\theta}(\iota) := \begin{cases} \theta(\iota) & \iota \in \text{dom}(\theta) \\ \sigma(\iota) & \iota \in \text{dom}(\sigma) \setminus \text{dom}(\theta) \\ \text{undefined} & \text{otherwise} \end{cases}$$

[5]The handling of undefinedness will be discussed in Sect. 3.1.2.

Likewise, we consider a partial function $\theta : \mathcal{U} \times \mathbb{I} \to \mathcal{U}$ and introduce an update h^θ on the heap.

$$h^\theta(o, \iota) = \begin{cases} \theta(o, \iota) & (o, \iota) \in \mathrm{dom}(\theta) \\ h(o, \iota) & \text{otherwise} \end{cases}$$

Call stack

To keep track of the overall program trace, we also introduce a *call stack* which stacks up the currently running methods. At first, we need to formalize the notation of a method.

Definition 2.8 (Method)**.** A *method* μ is formally given by a tuple $(C, m, \pi, \langle (T_1, \iota_1), \dots, (T_k, \iota_k) \rangle, T_R)$ where

- $C \sqsubseteq$ `Object` is a class,
- m is an identifier which denotes a (possibly constructor) method which is defined C,
- π is code fragment, called the *method body*,
- (T_j, ι_j) are pairs of types and identifiers given as parameters of m (the set may be empty),
- $T_R \in \mathcal{T} \cup \{\text{void}\}$ is the return type.

A non-static method defined in class C may be *called* on an object $o \in V_C^-$. In this case, o is called the *receiver* For a constructor, we define the object to initialize as the receiver. Static methods lack a receiver.

In Java, there may be several method declarations in the same class with the same method identifier. This is known as *method overloading*. They are distinguished by their *signature* which is made up of the identifier and the sequence of parameter types. Return types of methods with common signatures are required to equal in Java.[6] These methods are always considered different both in Java and JML.

In addition, non-static methods declared in super-classes can be *overridden* in sub-classes by methods with the same signature. These methods are also considered different, but share a common appearance. In different contexts, the same method call (which is given by the signature) thus leads to invocations of different methods. In this case, known as *dynamic dispatch*,

[6]This is at least true for Java 1.4, which we assume throughout this work. See Appendix B for a discussion on Java revisions.

the chosen method depends on the dynamic type of the receiver object. The *most-specific method* is then taken from the least class (according to the given type hierarchy) which contains a method declaration with that signature.

Let $\mathfrak{s}$ be a method signature and $M(\mathfrak{s})$ the set of methods with that signature. Since Java only allows one method with the same signature to appear in a class, $M(\mathfrak{s})$ can unambiguously be partially ordered by $\sqsubseteq$. For a given signature $\mathfrak{s}$ and receiver object $o \in V_T^0$, for which at least one method of that signature is declared in a super-class of T, the most specific method is then given by:

$$\mathrm{msp}(\mathfrak{s}, o) = \min_{\sqsubseteq}\{\mu \in M(\mathfrak{s}) \mid \mu = (T', \dots), T \sqsubseteq T'\}$$

Definition 2.9 (Call stack)**.** Let C denote the set consisting of all static methods of program Π and pairs (μ, r) of non-static methods or constructors and receivers $r \in V_{\texttt{Object}}$.

A *call stack* χ is a partial function $\chi : \mathbb{N}_+ \nrightarrow C$ such that there is a $k \in \mathbb{N}$ with $\operatorname{dom}\chi = [1, k]$. (It is $\operatorname{dom}\chi = \emptyset$ for $k = 0$.) Let X denote the set of call stacks. We define the following *stack operations*:

1. push : $X \times C \to X$ with

$$\mathrm{push}(\chi, c)(n) = \begin{cases} \chi(n) & n \leq |\chi| \\ c & n = |\chi| + 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

2. top : $X \nrightarrow C$ with

$$\mathrm{top}(\chi) = \begin{cases} \chi(|\chi|) & |\chi| > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

3. pop : $X \to X$ with

$$\mathrm{pop}(\chi)(n) = \begin{cases} \chi(n) & n < |\chi| \\ \text{undefined} & \text{otherwise} \end{cases}$$

We also overload the set inclusion relation $\in$ in that we write $c \in \chi$ for $\exists n \in [1, |\chi|] : \chi(n) = c$. A call stack represents the state of a program flow. In particular, there is an element $\mathrm{top}(\chi)$ referencing the method currently in progress.

System states

The triple (h, σ, χ) of Heap and Stack and call stack will be called a *system state* or simply a *state*. The set of all system states of a program Π is denoted by $\mathcal{S}$. For a partial function $\theta\colon \mathbb{I} \cup \mathcal{L} \nrightarrow \mathcal{U}$ we write $s^\theta := (h^\theta, \sigma^\theta, \chi)$ for short. (It should be clear to distinguish which values are on the heap and which on the stack.)

It will later become necessary (or at least very helpful) to restrict states to some sort of reachability. To assume a "global pre-state", which corresponds to an invocation of the `main` method of a program, is too restrictive for our more modular method specifications. When later dealing with method specifications, we will consider all theoretically possible pre-states, not just those which appear in an actual program execution. The following definition thus describes all states which are reachable by *some* program. It basically postulates that we only have to deal with created objects and there are only finitely many of them.

Definition 2.10 (Reachable state)**.** Let $s = (h, \sigma, \chi) \in \mathcal{S}$ be a system state. s is *reachable* if

- The call stack is non-empty: $\chi \neq \emptyset$,
- There are only finitely many created objects:
$$|\{o \in V_{\texttt{Object}} \mid h(o, \texttt{\textbackslash created}) = \mathit{true}\}| < \infty$$
- Every receiver on the call stack is created:
$$(\mu, o) \in \chi \Rightarrow h(o, \texttt{\textbackslash created}) = \mathit{true}$$
- Every object reference on the Stack is either created or null:
$$\sigma(\iota) \in V^-_{\texttt{Object}} \Rightarrow h(\sigma(\iota), \texttt{\textbackslash created}) = \mathit{true}$$
- Created objects only refer to created objects:
$$\begin{aligned} & o \in V^-_{\texttt{Object}} \wedge h(o, \texttt{\textbackslash created}) = \mathit{true} \Rightarrow \\ & \qquad \forall x \in \mathbb{I}.((o, x) \in \mathcal{L} \wedge h(o, x) \in V^-_{\texttt{Object}} \Rightarrow \\ & \qquad\qquad h(h(o, x), \texttt{\textbackslash created}) = \mathit{true}) \end{aligned}$$
- All classes and interfaces have been successfully loaded and initialized.

In the following, we will require a pre-state of a program execution to be reachable. In that way, it is guaranteed for the code to invoke an expected behavior. E.g. if the above set X is restrained to call stacks of reachable states, the stack operation top becomes a total function.

2.4 Java – a Black Box

We are now able to describe the execution behaviour of a Java code fragment through a black box using a Java Virtual Machine (JVM) [LY99, SSB01]. Although we take no inside view of neither the program code nor how it is processed, we expect every change of the memory state to be observable (to the extend of how we defined the state). Therefore, the black box yields a complete sequence of system states through which the program flow has run.

We further expect the machine not to throw any unexpected errors[7], i.e. errors which are not explicitly or implicitly (e.g. by assertions in Java[8]) declared in the given code fragment. E.g. we wish the machine not to run out of memory, while an explicitely thrown `new OutOfMemoryError()` may occur within the program code. Allowing such behavior would contradict a deterministic execution. Nevertheless, exceptions may be thrown at any time without any further restriction.

2.4.1 Object creation

As we have discussed before in Sect. 2.2, objects are not really created nor are they destroyed in any way. This approach is widely used in formal verification, especially with dynamic logic, where this feature is called the *constant domain assumption* [Har84, BHS07]. Since all objects exist (from a set-theoretical view) a-priori, we have introduced an implicit final field `\created` of type boolean to domain elements of type `Object` to distinguish created from not yet created objects.

The process of “creating” new instances can be broken down into three steps:

1. Creation: The `new` command is invoked for a type T. This causes an unused (i.e. $h(o, \texttt{\textbackslash created}) = \mathit{false}$) element o to be (deterministically[9]) fetched from V_T^0. The location $(o, \texttt{\textbackslash created})$ evaluates to *true* in the post-state.

2. Preparation: To all (non-static) fields defined in T and its super-types their default values are assigned (according to [GJSB00], see Table 2.3).

[7]Java defines any instance of `Throwable` or its subclasses to be an error unless it is an instance of `Exception` or its subclasses [Sun04].

[8]Not to be confused with JML's assertions (Sect. 4.3.2)!

[9]This can be done with an implicit counter on the domain elements as it has been done with `<nextToCreate>` in [BHS07].

3. Initialization: A constructor (of T or a super-class) is invoked. After the constructor invocation has finished (possibly abruptly) no field declared as *final* may be changed.

The reason for this quite complicated construction is that even if an object is not yet initialized – e.g. because the constructor terminates abruptly throwing an exception – it is already created and thus invariants are required to hold from now on. Furthermore, the reason why `\created` is given as a field rather than a predicate of some "higher logic" is that it has to be included in the definition of a system state. It could be thought of `\created` as a final ghost field in class `Object`, but since it is used slightly different from other ghosts, it is mentioned separately. Firstly, it has to denote a location even for those non-Java array types like `\bigint[]`, which are not subtypes of `Object`. And secondly, usual fields do not have values as long as the object is not created. Thus it cannot take the value $false$ prior to being created. We would have to mention explicitly which value `\created` holds and postulate that it is always legal to access this value.

Type T	Default value $d(T) \in V_T$
`boolean`	$false$
`\bigint`, `long`, `int`, `short`, `byte`	0
`char`	`u0000`
`\real`, `double`, `float`	0
reference or array type	null
`\TYPE`	null

Table 2.3: Default values for types.

Although the reference manual mentions the need of explicit constraining the state of an object on some occasion, e.g. regarding the range of quantification, yet there is no possibility to do so (like the use of the `\created` field).

Static initialization will not be considered in this work (see Appendix B). We expect all static fields to be initialized in every system state.

2.4.2 The black box

A (possibly infinite) sequence of states reached throughout the execution of a code fragment is called a *run*. A run thus can be seen as a partial function $\mathbb{N} \nrightarrow \mathcal{S}$ which is either total or, there is a $k \in \mathbb{N}$ such that every $n \leq k$ is in the domain and every $m > k$ is not in the domain. This is important not only to tell two states apart from each other, but also recognize the position of a certain state within the run. E.g. the pre- and post-state of a program fragment `{x = x;}` are identical, but there have been read and write accesses to the location of x. From a set-theoretical view, an element of a run would be an ordered pair $(n, s) \in \mathbb{N} \times \mathcal{S}$. Since this looks too monstrous for most uses, we carefully identify this pair with the state s. To retain at least a little precision, we will speak of a *state of the run* $s \in R$. We distinguish states of the run through indices. E.g. we say that a state s_i appears before a state s_j if $i < j$, while it may still occur that those states coincide.

Definition 2.11 (JVM black box)**.** Let $s \in \mathcal{S}$ be a reachable system state and π^A an annotated code fragment. A Java Virtual Machine *black box* execution is represented by a function $\mathcal{J}$ with $\mathcal{J}(s, \pi^A) = (R, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$ where

- R is the run of π executed in state s. R contains at least two states (pre- and post-state) for every statement which requires a memory operation of the underlying virtual machine.[10] In particular, s appears as the 0th element of R. It is of infinite length if and only if π does not terminate. In any other case, there is a particular last element (the post-state).
- $\lambda_a, \lambda_w : \mathbb{N}^2 \to 2^{\mathcal{L}}$ map pairs of states represented by their position in the run to the sets of accessed (read) and respectively assigned (written) locations between those states.
- $\alpha : R \to (\mathbb{N} \nrightarrow A)$ maps states of the run to sequences of annotations (see Sect. 4.3). For a loop annotation, the state after the loop condition has been evaluated is mapped to the annotation. For any other annotation, it is the state reached after the evaluation of the preceding Java statement.
- $\Omega \in V_{\texttt{Throwable}}$ indicates the mode of termination (if any): normal or abrupt termination by a throwable object, i.e. an exception or an error. A program fragment terminates *normally* if, execution reached the end of the statement block or it is terminated abruptly without the throw of a throwable object, i.e. by one of the built-in statements `break`, `continue` or `return`. Normal termination is signaled by $\Omega = \text{null}$.[11]

[10] This may include loading of constant values. Consider the code fragment `while (true);`. There is neither access on the Heap nor on the Stack. We however expect it to yield a run of infinite length.

[11] It is not possible that a method is terminated exceptionally through the throw of (the exception) null. If the throw of null is declared, the program terminates exceptionally with an instance of `NullPointerException` [LPC+08, Sect. 14.17].

- $\rho \in \mathcal{U}$ is a return value. Nothing is said in the cases where $\Omega \neq \text{null}$ or the return type of π is `void`. If π is the body of a constructor, then ρ is the initialized object.

The black box execution follows the Java specification [GJSB00, LY99]. It particularly ensures that:

- It is executed deterministically and no unexpected error is thrown.
- Objects are created as given above.

Three important observations follow from this definition: First of all, every state of the run is reachable. Secondly, the values of final fields never change after the constructor invocation is finished. I.e. for any state (h, σ, χ) which occurs in any sequence after the post-state (h_0, σ_0, χ_0) of a constructor invocation for an object o (or o has already been initialized in the pre-state) it is $h(o, x) = h_0(o, x)$ for a final field x. This particularly includes `\created`. And finally, upon a method invocation the method/receiver-pair is put on the call stack and the values of all local variables (including method parameters) are defined through the stack description. After the method invocation is finished the before values of all parameters are restored.[12]

Example 2.12. Consider this short code fragment calculating the factorial of some integer value stored in x:

- $h \subseteq \{(o, \texttt{\textbackslash created}) \mapsto \textit{true}, (o, x) \mapsto 10\}$, $\sigma = \{y \mapsto 1\}$ and $\text{top}(\chi) = \{(\mu, o)\}$
- $\pi^A :=$ `while (x > 0) { y = y * x--; }`

Here, o is the only created object, (o, x) the only location of interest and y a local variable. Then, the black box calculates

- $R = \langle s_0 = (h, \sigma, \chi), \ldots, s_k = (h', \sigma', \chi') \rangle$
- $\lambda_a(0, k) = \lambda_w(0, k) = \{(o, x)\}$
- $\alpha = \{s \mapsto \emptyset \mid s \in R\}$ and $\Omega = \text{null}$

with $h' \subseteq \{(o, \texttt{\textbackslash created}) \mapsto \textit{true}, (o, x) \mapsto 0\}$, $\sigma' = \{y \mapsto 3628800\}$ and $\chi' = \chi$.

[12]This is important not only because JML does always refer to the values of parameters in the pre-state (see Sect. 4.2), but also in order for the black box to handle recursion correctly.

In the above definition, we omitted giving an exact measure of the granularity of R. The reason for this is, that is not trivial to declare a one-to-one relation of source code and states. While it may appear that some lines of code do not have any effect on the internal state of the machine, on the other hand it may well be that one single assignment requires several steps. As we wish to record every access (reading or writing) on the Heap, even a self-assignment such as `a.x = a.x;` does require at least two elements to appear in the run. Therein, it is of no importance that those two represent the very same state, but that access on the Heap occurred.

It also does not matter *how many* intermediate states are contained within the run. One could for instance think of 1 for loading the local variable `a`, 2 for loading the field `x` and 3 for the assignment – or 5 since `a.x` needs to be loaded twice, or maybe even more. The question of how many memory accesses take place is none of the syntax of a program, but of a compiler which transforms the program to machine code or an intermediate (e.g. Java bytecode). Therefore, we used the terminology "required memory operation" to signify that *any* Java compiler would instruct the machine to take such an operation. As a result, we do not have a lower boundary of granularity.

2.5 Privacy and inheritance

Privacy modifiers

In JML, not only classes and their members may be prefixed with a privacy modifier, but also specifications such as invariants or behavioral cases in a method specification. These modifiers are the same as in Java: `private`, `protected`, default (package visibility), `public`.

It has been argued [Mey97] that a specification should not refer to fields, methods, classes etc. which are more private than itself. E.g. a public invariant may only constrain public fields. We do not question this approach so far and assume every specification to be syntactically valid which includes these access restrictions. To work around those, one can alter privacy modifications given in Java via JML's `spec_public` and `spec_protected` modifiers, for instance:

```
        private /*@ spec_public @*/ int z;
```

To a Java program z is still private while in a specification context it may be accessed publicly.

Privacy modifiers are mostly a syntactical feature, and access rules are therefore ignored in this work. The only difference they impose on semantics, whether specifications of super-types are inherited by sub-types.

Inheritance of specifications

Specification elements such as invariants, history constraints or method specifications defined in some type are inherited by its sub-types. This principle follows Java privacy rules for inheritance: A specification is inherited if and only if its visibility is at least `protected`, or it has default visibility and both super- and sub-type are contained in the same package.

This means for a method in a class C to respect all invariants and history constraints of super-classes of C, as well to fulfill the method contracts of every method which it overrides. (Note that there may exist methods with the same name but different parameter types which are not overridden, but overloaded, and thus (at least in a formal sense) unaffiliated.) As a consequence, specifications which are added in the sub-class must not be any weaker than the inherited specifications, effectively. Particularly, within a method specification, preconditions may only be weakened, and the list of locations of a frame condition (such as `only_assigns`) may only be amended.

The latter gives rise to a particular problem since specifications in super-classes have no way to refer to fields defined in sub-classes. JML's solution to this is the introduction of *data groups* (see Sect. 3.2). An example can be found in Fig. 3.3. This example also shows that there exist different concepts about the term "sub-type". JML semantics enforce sub-typing always to be *behavioral* [LW94]. But this discussion will not be part of this work.

The meaning of all inherited specifications is typically captured into a single clause or expression, e.g. inherited preconditions of a method specifications are conjoined together. Since expressions are linear by nature, the type hierarchy is not. Therefore, we have to define a linear ordering of those specifications, i.e. a linear ordering on types which is compliant with the given type hierarchy.

Definition 2.13. A binary relation $\preceq$ is a *linear type hierarchy* if

(i) it is a total ordering on $\mathcal{T}$ and

(ii) it preserves the type hierarchy: $A \sqsubseteq B \Rightarrow A \preceq B$

Since the Java/JML type hierarchy is directed and acyclic, there always exists a linear form.Linear type hierarchies can for instance be found by breadth-first-searches starting in `java.lang.Object`. The position of all other types (i.e. primitive and JML types) is unimportant and thus may be chosen arbitrarily. For example the types given in Fig. 2.1 can be ordered linearly as `Object` $\succeq$ `Comparable` $\succeq$ `int[]` $\succeq$ `Object[]` $\succeq$ `String` $\succeq$ `Comparable[]` $\succeq$ `Object[][]` $\succeq$ `String[]` $\succeq$ `int` $\succeq$ `\TYPE` $\succeq$ `\TYPE[]`.

Chapter 3

Expressions in JML

This chapter discusses the values of expressions in JML. Some of them are directly taken from Java, such as numerical expressions and most boolean expressions. There are additional boolean operators in JML, such as `==>` (forward implication), `<==` (backward implication), `<==>` (equivalence) and `<!=>` (antivalence). Those are only short-hand for existing logical operations, while quantifiers `\forall` and `\exists` significantly enhance the expressiveness of the language. JML also allows "side-effect free" methods to be used like mathematical functions. In addition, there are expressions which are exclusively used in postconditions of method specifications and history constraints. Firstly, there are predicates for the specification of frame properties, e.g. `\not_assigned` names those locations which must not have been assigned to. Secondly, the operator `\old` allows one to refer to the pre-state value of arbitrary expressions.

We will denote the sub-language of expressions which are allowed to appear anywhere, e.g. in preconditions, and do not contain reference to model fields by JML-E_0. This means that those expressions must not contain keywords like `\old` which refer to another state. Thus, JML-E_0 can always be evaluated within one single state. The full set of expression will be denoted JML-E_1. The complete syntax definition of expressions is given in [LPC$^+$08, Sect. 11.3]. In this work, we only consider well-formed expressions according to this reference.

JML expressions are typed, such that we may refer to expressions of boolean, numerical, reference type, etc. By an expression *expr of type* T we mean that it is syntactically compatible with type T, i.e. *expr* has static type T. In the rare cases where we are interested in a specific dynamic type T we will determine it through the set of direct instances V_T^0. In particular, we will speak of expressions of boolean type rather than logical formulae. The definition of validity is thus based on boolean expressions which valuate to true. We denote the set of well-formed expressions by $\mathcal{E}$ and the subset of boolean expressions $\mathcal{E}_{\text{bool}}$.

In this chapter we will at first evolve a notion for the evaluation of expressions and a definition of validity for boolean expressions. In the beginning, we only take JML-E_0 expressions into account, which evaluation only depends on one system state. This can be seen as a "naïve approach" for evaluation. The more general case, in which it is both allowed to refer to the pre-state and to use model fields, is based on that first approach. Following that, Section 3.2 explains the notion of *data groups*, which define collections of locations to be used within frame conditions. In the final section, we will discuss the values of all covered JML expressions.

3.1 Evaluation function and validity definition

In this section we lay the grounds for the evaluation of expressions. Basically, we employ a valuation function *val* which maps expressions (of any type) to semantical elements of the universe, which we have defined in the preceding chapter.

Beneath that, we define a *well-definition* predicate *wd*, which asserts the semantically legal use of a (syntactically well-formed) expression in some context. In order for a boolean expression, i.e. the equivalent to a formula, to be *valid*, it has to valuate to *true* and to be well-defined.

Since expressions in general may have side-effects, we also have to discuss how to capture information on side-effects caused by certain expressions. Following that thought, we introduce a function ω which maps expressions to a state transition function.

The first approach to validity of expressions only takes expressions of the sub-language JML-E_0 into account. With this definition, "one-state properties" like preconditions and invariants which do not contain model fields can be evaluated. This is important for the succeeding definition of model field valuation, which is built upon that "primitive" first definition of validity. The second approach to validity thus incorporates both a way to refer to model field values as well as semantical support for the `\old` expression and frame conditions.

For a final definition of validity, *axioms* are taken into account, additionally. These can be seen as propositions which are "always true". This is equivalent to: If any axiom does not hold (in some context), then any other proposition becomes true trivially.

3.1.1 Valuation function

The valuation function $val : \mathcal{E} \rightarrow \mathcal{U}$ evaluates expressions in a compositional manner. The value of an expression thus depends on the values of its sub-expressions. Atomic expressions are constants, local variables and the keyword `this`.

Since tertium-non-datur holds for boolean expressions (at least on the semantical domain), we only provide the case in which it evaluates to *true*. E.g. "$val(a \texttt{ || } b) = true$ iff $val(a) = true$ or $val(b) = true$" is short-hand for the following definition:

$$val(a \texttt{ || } b) = \begin{cases} true & val(a) = true \text{ or } val(b) = true \\ false & \text{otherwise} \end{cases}$$

The evaluation of references depends on a system state in which the expression is evaluated. Local variables are valuated through the Stack and fields through the Heap. We denote the evaluation in state s by adding a subscript val_s.[13] Let for instance be $s = (h, \sigma, \chi)$. It is

$$val_s(x) = \begin{cases} \sigma(x) & x \text{ is a local variable}^{14} \\ h(val_s(\texttt{this}), x) & x \text{ is a field} \end{cases}$$

Since for some expressions, their value does only depend on their sub-expressions and not on the system state, we will partly omit naming it.

One fundamental problem is how to refer to semantical entities from some expression. Since we have defined the Heap function for the pair of a semantical object and an identifier, there has to be some point to begin evaluation. (In the preceding example we have inserted `this` to circumvent this problem.) A similar situation applies to method calls; in order to evaluate it, there has to be a semantical object present. This is easy to deduce if the first reference of a chain is a local variable, e.g. $a.b.c$ can be evaluated if a is a local variable. In the case it denotes a field, the expression has to be transformed in order to be evaluated.

Definition 3.1 (Normalized expression)**.** To an expression e there is a semantical equivalent expression $e^{\mathcal{N}}$ in which

- every non-static field or method identifier is prefixed with "`this.`"

[13]The state written in subscript can be seen as a first parameter of $val : \mathcal{S} \times \mathcal{E} \rightarrow \mathcal{U}$

[14]Remember that σ is defined as a partial function. Any identifier which is not a local variable is not in the domain of σ. In order for val to be a *total* function, we define the respective values as underspecified. (See also Sect. 3.1.2)

- every static field or method identifier of class C is prefixed with "C.". "`super`" is replaced accordingly.

We call this the *normalized expression*. It can easily be produced using (syntactical) substitutions.

In a normalized expression, a reference to a non-static field x is transformed into `this`.x. `this` is a special keyword representing the current receiver object, which is evaluated from the system state's call stack. Thus, normalized expressions are meant to be evaluated in a state in which `this` evaluates to the receiver.

3.1.2 Undefinedness

Throughout the evolution of both formal logic and computer programming, the issue on how to treat the undefined gave birth to several different approaches. Early logicians developed three-valued (or more general multi-valued) logics [Łuk20]. These include explicit truth values "undefined". Three-valued logics however may differ in the way they extend the operators of classical logic.

Underspecification versus exceptions

More recent approaches [GS95] avoid explicit undefined values by *underspecification*. The basic idea is to extend partial functions to total ones by adding well-defined but unknown result values to argument tuples which are not in the domain. E.g. there is some integer x satisfying $x = 23/0$. It is the responsibility of the specification to rule out that there will ever be an illegal operation like division by zero, e.g. $z = 0 \lor x = 23/z$. The great advantage with this approach is that axioms of classical logic, such as tertium-non-datur, are still valid. E.g. $x = 23/z \to x = 23/z$ still holds – even if z evaluates to zero. [Häh05] argues that this approach is superior in the context of specification. Other specification languages such as OCL also follow this approach [OCL05].

Modern programming languages like Java, on the other hand, usually deploy facilities such as errors or exceptions, which are thrown at runtime when execution is faced with undefined values. This is the only reasonable action on the program level, but it raises the question of how exceptions should be represented on the semantical level.

Initially, JML semantics for undefinedness were also based on underspecification [LCC$^+$05, Sect. 4], but as a major change in 2007, semantics are now based on “strong validity” [KTB91]. The JML reference manual [LPC+08, Sect. 2.7] defines this by:

> [A]n assertion is taken to be valid if and only if its interpretation does not cause an exception to be raised, and yields the value true.

This does in fact not cover the whole issue. JML features Java’s *short circuit evaluation* as a protective element: Earlier clauses (more left in the formula) can protect later ones against undefinedness. The example from above `z == 0 || x == 23/z` still holds since the exception is never thrown due to short circuit evaluation. Likewise, on the specification level preconditions may protect invariants and postconditions.

Although this approach deviates a long way from classical (two-valued, commutative) logic[15], it has been argued that it helps to locate programming errors more quickly, and it is better understood by programmers [Cha07]. In this work we will both use the underspecification technique for otherwise undefined values as well introduce a predicate which asserts well-definition of expressions, i.e. the absence of exception.

Well-definition predicate

As it has been explained in length above, JML boolean expressions do not form a standard two-valued logic. Therefore we need to introduce a *well-definition predicate wd* besides the valuation function *val*. This has been proposed in [Cha07]. It is primarily used to tell whether Java would throw an exception on evaluation of that expression. Therefore, we will mention the respective exception type, if this is applicable.

The well-definition function can be seen as a kind of a second, orthogonal evaluation function. On the top-level, a boolean expression is taken to be valid if and only if it evaluates to true and it is well-defined.

For all expressions except the short-circuit versions of boolean operators, *wd* is true if and only if *wd* holds for every subexpression. Consider for instance the short-circuit disjunction `||` again. The second operand is only evaluated if the first one evaluates to $false$. This leads to:

- $val(a \texttt{ || } b) = true$ **iff**
  - $val(a) = true$ and $wd(a)$ **or**
  - $val(b) = true$ and both $wd(a)$ and $wd(b)$.

[15]It may be seen as a “covered” three-values logic approach with an exceptional truth value.

- $wd(a \texttt{ || } b)$ **iff** $wd(a)$ and if $val(a) = false$ then also $wd(b)$.

This definition reveals the semantic difference between JML's two implication operators: (a `==>` b) is equivalent to (`!`a `||` b) whereas (a `<==` b) is equivalent to (a `||` `!`b). In the first case, b does not need to be well-defined if a is false, while in the second case b does not need to be well-defined if a is true.

The well-definition predicate is *not* used to tell whether an expression is syntactically well-formed. Syntactical correctness is always assumed. In particular, wd does not report Java expression which result into an error at *compile time* according to [GJSB00], but exceptions which would be risen at *run-time*. E.g. `null != 42` is not a syntactically valid expression since the inequality operator is typed in Java. (And thus in JML.) However, it would be well-defined (and valuate to true) in our formalization.

Like the valuation function, wd also depends on the state of evaluation. Consider for instance a field access through an expression $a.x$ where a is an expression of reference type and x denotes a field. With $s = (h, \sigma, \chi)$, valuation is done by $val_s(a.x) = h(val_s(a), x)$. But this expression is only well-defined if a is not a null reference. (This corresponds to the throw of a `NullPointerException` in Java.) Thus, a necessary condition for $wd_s(a.x)$ is $val_s(a) \neq \text{null}$.

3.1.3 Expressions with side-effects

In JML, expressions are not necessarily completely free of side-effects. This means that subexpressions may be evaluated in different states. JML does however restrict expressions to be *pure*, which can be seen as "weakly side-effect free".

Pure methods

It has been found very helpful for specifications to make use of program methods which behave similarly to mathematical functions. For this reason, in JML methods can be declared *pure*. This means they must terminate (normally or exceptionally) in any case and may not change locations which have existed in the pre-state, i.e. they are (to some extent) side-effect free.

They also have a trivially true pre-condition, so that they may be called in any context. Pure methods might however throw exceptions and as a consequence not always deliver a valid return value. In combination with the definition of undefinedness (see Sect. 3.1.2), this definitely makes sense since parameters of a pure method should not be restricted to values for which exists a result.

According to the JML reference manual [LPC+08], one can rely on all pure methods of Java standard libraries being already declared pure. Classes may also be declared pure which declares every method to be pure (those of sub-classes too). [SR05] covers methods of purity analysis for Java methods. But that is not part of this work.

Weak purity

Constructors may also be declared pure. In this case, they are only allowed to assign fields of the instance they create (referred to as `this`). This means however, that the post-state of a pure method (which may have called pure constructors) does not necessarily equal its pre-state [BNSS05]. Such a method is called *weakly pure* [Cok05, DM06].[16] In this context, some authors regard it as "side-effect free", since it does not assign locations which were present in the pre-state.

From a practitioner's point of view, this is a very common and accepted technique. By the principles of object-orientation, any data which represents more than a single numerical value is represented as an object. E.g. in Java, a `String` is *not* a sequence of characters, but an object which needs to be created. As another example, Java does not allow an element to be appended to an existing array (since it has fixed length), but to copy its contents to a newly created array. In addition, methods often create temporary objects such as iterators.

From a theoretical view however, this raises the question, in which state an expression containing calls to weakly pure methods is evaluated and how it does affect later computations. The JML reference manual does not mention this at all. We base our interpretation on three principles:

1. Specification expressions must not interfere with computation. Therefore, the scope of an altered system state is bounded to expression evaluation. Computation continues from the original state.

2. In order to conform with overall JML policy, the behaviour of expressions should resemble their counterparts in Java. In particular, the *values* of new objects should be as expected. E.g. `new Integer(3).equals(new Integer(3))` should hold, but `new Integer(3) == new Integer(3)` is not necessarily true. It is also desirable to imitate the linear order of evaluation.

[16] In this way, pure methods in JML do differ from *queries* in UML [RJB99], which are required to be absolutely free of side-effects.

3. After all, semantics have to be well-defined. It should for instance be decidable within finite time whether `(\forall Object x; true; x == new Object())` holds. (It should certainly not.) Therefore, the interpretation may only change the state of evaluation a finite number of times.

[DM06] presents a solution based on *pure successor* states. In this, every specification expression does not only yield a value, but also a transition between states. In this way, the left-most subexpression yields a successor state which serves as the state of evaluation for the second left-most subexpression, and so on.

ω function

The pure successor function $\omega : \mathcal{E} \to (\mathcal{S} \to \mathcal{S})$ assigns a state transition operator to every expression. It is ineffective on most expressions, however. Only creation of new objects (which may be invoked by weakly pure methods) leads to transitions from the original state of evaluation to a *pure successor state*. In general, the resulting transition function is just the composition of the transition functions of all sub-expressions. Let us consider the predicate `\fresh` applied on n expressions:

$$\omega(\texttt{\textbackslash fresh}(e_1, \dots, e_n)) = \omega(e_n) \circ \omega(e_{n-1}) \circ \dots \circ \omega(e_1)$$

As we have discussed above, these transitions should resemble those of Java expressions. Thus, for the short-circuit logical operators, only those sub-expressions which are evaluated may produce a pure successor:

$$\omega(a \texttt{ || } b)(s) = \begin{cases} \omega(a)(s) & val_s(a) = true \\ \omega(b)(\omega(a)(s)) & val_s(a) = false \end{cases}$$

The major disadvantage of this approach is that the order of subexpressions does *always* matter, e.g. a simple numerical addition will not be commutative anymore:

$$val_s(\texttt{a + b}) = val_s(\texttt{a}) +_{\texttt{int}} val_{\omega(\texttt{a})(s)}(\texttt{b}) \neq val_s(\texttt{b + a})$$

As we will later show in detail, a successor of a reachable state is always reachable, too.

3.1.4 A first definition of validity

As we have discussed above, a boolean expression is (semantically) valid on the top-level if it valuates to *true* and it is well-defined. Putting this in a formal definition leads us to the following:

Definition 3.2 (Validity of expressions – 1st approach)**.** Let ϕ be a normalized expression of type boolean and $s \in \mathcal{S}$ be a state. ϕ is *valid* in s iff $val_s(\phi) = \mathit{true}$ and $wd_s(\phi)$ holds. We write $s \vDash \phi$.

This definition resembles in a way truth definitions of modal logics.[17] And, it is sufficient for the evaluation of the sub-language JML-E$_0$. Several features of JML however, force us to further extend this definition.

First of all, the `\old` operator forces us to reason about two different states at a time. Some approaches try to master this by application of substitutions (see for instance [BBS01, Eng05]). We will however pursue a more elegant approach involving valuation with two states. Furthermore, reasoning about method specifications (Sect. 4.2) will require to take result values and accessed/assigned locations into account. We extend val to $val_{(s_0,s_1,L_a,L_w,M,\rho)}$ (and wd alike) where s_1 is the "current" state and s_0 is the "old" state, L_a and $L_w \subseteq \mathcal{L}$ are sets of accessible resp. assignable locations and $\rho \in \mathcal{U}$ the return value. $M : \mathcal{S} \times \mathcal{U} \times \mathbb{I} \to \mathcal{U}$ is a special valuation function, which will be explained in the up-following subsection. We will call the tuple $\Sigma = (s_0, s_1, L_a, L_w, M, \rho)$ a *logical state* (as opposed to a system state).[18] This leads to the following definition:

Definition 3.3 (Validity of expressions – 2nd approach)**.** Let ϕ be a normalized expression of type boolean and Σ a logical state. ϕ is *valid* in Σ iff $val_\Sigma(\phi) = \mathit{true}$ and $wd_\Sigma(\phi)$ holds. We write $\Sigma \vDash \phi$.

If the evaluation of an expression does not depend on some part of the logical state, it will be partly omitted (e.g. val, val_s or $val_{(s_0,s_1)}$ may occur). This means for all parts not mentioned to be passed to subexpressions without further notice. E.g. we write $val_s(\texttt{n + m}) = val_s(\texttt{n}) + val_{\omega(\texttt{n})(s)}(\texttt{m})$ as short-hand for $val_{(s',s,L_a,L_w,M,\rho)}(\texttt{n + m}) = val_{(s',s,L_a,L_w,M,\rho)}(\texttt{n}) + val_{(s',\omega(\texttt{n})(s),L_a,L_w,M,\rho)}(\texttt{m})$. Likewise, if the validity of an expression *must* not depend on the whole tuple, we will write $s \vDash \phi$ for short. This means for all parts not mentioned to be arbitrary.

3.1.5 Model fields

One feature of JML annotations to Java programs is that additional fields may be added for the purpose of abstraction. As with all other features,

[17]In opposition to most formal logics, we completely omitted a reference to a kind of logical signature or at least the universe of domain elements. This is purely for convenience reasons. Everything we need as structural foundations is present in the program Π which we fixated for our considerations, such as the universe $\mathcal{U}$. (See Sect. 2.1)

[18]With the evaluation in a logical state, val has effectively signature $\mathcal{S} \times \mathcal{S} \times 2^{\mathcal{L}} \times 2^{\mathcal{L}} \times \{\mathcal{S} \times \mathcal{U} \times \mathbb{I} \to \mathcal{U}\} \times \mathcal{U} \times \mathcal{E} \to \mathcal{U}$.

those *model fields* [CLSE05] are not part of a program and are only used throughout specification. Model fields are declared similarly to regular fields, but within comments, e.g. through `//@ public model int z;`. Model fields may have any JML type, including `\TYPE`. Model fields may be declared in both interfaces and abstract classes.

In contrast to regular fields, model fields are not directly assigned (by neither the program nor assignments on the specification level), but they are meant to represent some abstraction from concrete values. Therefore, we explicitly excluded model fields from the definition of locations (see Sect. 2.3). Values of model fields are specified using the `represents` statement. It comes in two flavours:

- Functional abstraction: The syntax is as for an assignment to the model field m with a compatible expression e on the right-hand side. It means that in every state the values of m and e must be equal.
- Relational abstraction: A boolean expression b, the *representation clause*, is given, which must evaluate to true in every state. This is defined with the keyword pair `represents`/`\such_that`.

It can be easily seen that the former is just a special case of the latter with $b =$ (`m == e`). For simplicity we will only consider relational abstractions. These can be thought of as strong invariants, i.e. they are meant to hold in every state.[19] The following example defines an integer z which has no concrete value but is constrained to be a positive even number.

```
/*@ public model int z;
  @ represents z \such_that z > 0 && z%2 == 0;
  @*/
```

Representation clauses are in general defined with the following specification:

```
//@ M represents x \such_that ψ;
```

Where M is a privacy modifier. The representation clause ψ is a boolean expression. Since the model field may be inherited, there may be additional representation clauses declared in subtypes. If a model field is declared in type T and we consider an instance $o \in V_{T'}$ with $T' \sqsubseteq T$ as receiver for

[19]The JML reference manual postulated a weaker condition, such that these representation clauses only have to be met in certain states, namely visible states. This design decision is primary made under certain assumptions on observability and enforceability, which are unimportant to our theoretical considerations. Therefore, our viewpoint is teleologically justified.

this field, the `public` and `protected` representation clauses of all types T'' with $T' \sqsubseteq T'' \sqsubseteq T$, as well as all representation clauses with default privacy of all types T''' with $\{T''', T\} \subseteq P \in \mathcal{P}$ and $T' \sqsubseteq T''' \sqsubseteq T$ apply. Let Ψ be the set of all applicable representation clauses. Since any type contains at most one representation clause, Ψ can be ordered totally by some linear type hierarchy (see Def. 2.13). Let $\langle \psi_1, \psi_2, \ldots, \psi_k \rangle$ be the resulting ordered sequence of normalized representation clauses from Ψ. We denote the short-circuit conjunction by $\psi^T_{T'} := \psi_1$ `&&` ψ_2 `&&` $\ldots$ `&&` ψ_k, and define $\psi^T_{T'} :=$ `true` in the case $\Psi = \emptyset$.

Evaluation of model fields

So far, the *values* of model fields have not yet been taken into account. The main reason for this is, that they can not be valuated the straight way. As it has been discussed above, model fields only represent an *abstract* value. The issue of handling model fields is still subject to on-going research. As an early approach, it has been proposed in [Mül02] to regard model fields as a sort of pure method, which is evaluated when needed. This however covers only the special case of functional abstraction.

Several approaches [BP03, Eng05, LM06, DM06] try to incorporate the more general case of relational abstraction in which the model field may not have a concrete value. Model fields and their representation clauses are also inherited; where additional representation clauses may be added by the inheriting class. In addition, model fields may depend on other model fields. This fact might turn the problem more complicated. Let us lay some common ground for reasoning about model fields.

Definition 3.4 (Model field characterization)**.** Every model field instance is characterized by a tuple $(o, x, \bar{\psi}) \in \mathcal{U} \times \mathbb{I} \times \mathcal{E}_{\text{bool}}$ where

- $o \in V^0_{T'}$ is the receiver of the (possibly static) field[20], or $o = \text{null}$,
- x is the identifier of the field declared in type T and
- $\bar{\psi} := \psi^T_{T'}$ is the conjunction of all applicable representation clauses. If $o = \mathit{null}$ it is $\bar{\psi} := \psi^T_T$.

The set of all these tuples (for all possible receivers) is denoted by $\mathfrak{M}$.

As for concrete fields, there is a $(o, x, \bar{\psi})$ to a static field x for every $o \in V_T$. It is vital to have the null object included in this definition. Since model fields are allowed to appear as members of interfaces, there may be no

[20] o is not necessarily created.

other instance of that type. (Consider for instance an interface with static fields, but it is never implemented by a non-abstract class.)

A quantification-based approach

[BP03] presents two approaches using quantification, which are widely used. We will discuss them in short and then point out why they are insufficient. Every expression ϕ is transformed in a way that it appears as the body of a quantifier expression with the model field as the variable to be quantified over. The clear advantage of this technique is, that everything can be determined on the expression level. The first of those two approaches uses existential quantification:

$$\phi \quad \rightsquigarrow \quad (\texttt{\textbackslash exists}\ T\ x; \bar{\psi}; \phi)$$

This transformation takes place recursively for every element of $\mathfrak{M}$. However, this condition is clearly too weak since it only asserts the existence of *one* value satisfying the representation clause.

The second approach builds on both universal and existential quantification. It does both assert ϕ under the precondition that $\bar{\psi}$ holds, as well as the existence of such a value. (Universal quantification alone would mean any expression to be trivially true if one representation clause is unsatisfiable.)

$$\phi \quad \rightsquigarrow \quad (\texttt{\textbackslash forall}\ T\ x; \bar{\psi}; \phi)\ \texttt{\&\&}\ (\texttt{\textbackslash exists}\ T\ x; \texttt{true}; \bar{\psi})$$

This approach still has got some major flaws. First of all, the order in which the expression is transformed, i.e. an ordering imposed on $\mathfrak{M}$, does matter. Model fields may depend on each other, so if $\bar{\psi}$ contains reference to another model field y, the scope of quantification of y has to include $\bar{\psi}$. This would mean the semantics of model fields only to be well-defined if there exists a linear ordering of dependencies in $\mathfrak{M}$. This not always given: Consider a case in which the representation clauses of $\mathfrak{M}$ are declared to produce cyclic dependencies.

Secondly, in the presence of the `\old` operator, these substitutions do not produce well-formed expressions. Consider a model field identifier bound by `\old`. It cannot be bound by quantifiers, too. In this case, there is need for additional substitutions such as storing old values in separate model fields.[21]

The most severe problem with this approach is that it only talks about *all values* which satisfy the representation clause simultaneously. But there is no way to tell how *one concrete* value would behave. In particular, assuming concrete values – or more generally, strengthening the representation clause –

[21]In [Eng05], which also takes on this approach, this problem is avoided as the substitutions do not take place on the JML expression level, but the level of dynamic logic.

always leads to falsity. Consider the example in Fig. 3.1. Obviously, the two preconditions together cover every concrete value of x. But neither one holds *itself* for every concrete value, because (`\forall int x; x==0 || x==1; x==`a) is unsatisfiable for any a.

```
public class Model {
    //@ public model int x;
    //@ represents x \such_that x == 0 || x == 1;

    /*@ public normal_behavior
      @     requires x == 0;
      @     ensures  false;
      @ also
      @ public normal_behavior
      @     requires x == 1;
      @     ensures  false;
      @*/
    public void foo ();
}
```

Figure 3.1: Example for model field values based on quantification. The precondition becomes unsatisfiable when transformed according to the quantification-based approach.

This is not what we wanted. Intuitively, a model field is not supposed to be yet another way to impose invariants, but to represent some concrete value. As it turns out, any approach which builds on substitution on the expression level is insufficient. Therefore, in the following subsection, we will present an approach which works on a "higher level".

An approach with concrete values

Our own approach is based on the idea that model fields *do* denote concrete values, just as concrete fields do. [LM06] follows a similar thought and also stores model field values on the Heap. We however separate concrete and model fields, as to keep the program and its specification apart. Therefore, we introduce a function M called *model field valuator* which assigns a concrete value depending on the state of evaluation. The notion of locations is extended to include *abstract locations*:

$$\mathcal{L}' := \mathcal{L} \cup \{\ell \in \mathcal{U} \times \mathbb{I} \mid (\ell, \psi) \in \mathfrak{M}\}$$

In the following definition, we describe model field valuators by imitating concrete locations and their valuation through the Heap. That is, we extend the Heap of state s through the addition of model fields, such that every abstract location ℓ is mapped to a value denoted by $M(s, \ell)$.

Definition 3.5. Let $\mathfrak{M}$ be the set of model field characterizations as defined above. The set $\mathcal{M}$ is *either* the least set containing all functions $M : \mathcal{S} \times \mathcal{L}' \to \mathcal{U}$ such that for every system state $s \in \mathcal{S}$ and every $(o, x, \psi) \in \mathfrak{M}$ it is

- $M(s, o, x) \in V_{typeof(x)}$,
- $val_{s'}(\hat{\psi}) = true$[22] and
- $wd_{s'}(\hat{\psi})$ holds,

where s' coincides with $s^{\{\ell \mapsto M(s,\ell) \mid (\ell,\psi') \in \mathfrak{M}\}}$ except for $val_{s'}(\texttt{this}) = o$ and

$$\hat{\psi} := \begin{cases} \psi & \text{for static } x \\ \texttt{this.\textbackslash created ==>}\ \psi & \text{otherwise} \end{cases}$$

or, if this set is empty, then $\mathcal{M} := \{\emptyset\}$.

In the former case, $\mathcal{M}$ contains every valuation function which conforms to any representation clause at the same time. This only applies to created objects since otherwise concrete fields on which the model field depends may not contain specified values. In the latter case, $\emptyset$ can be seen as a partial function with signature $\mathcal{S} \times \mathcal{L}' \nrightarrow \mathcal{U}$ which is undefined for any parameter. The dependency problem discussed above does not occur in this approach since valuators "directly" assign concrete values to all model fields in parallel.

Taking model fields into account, specifications are always meant to hold for every possible valuator. E.g. the intuitive meaning of a method contract is: For every model field valuator and every system state in which the precondition holds, if the program terminates normally, then in the post-state the postcondition holds. In this approach, the interpretation of model fields is not local to one particular state, but to the whole specification. By this, we mean that in the pre-state, in the post-state and in any single one in between, model fields are valuated with one and the same valuator function M. In this way it is very easy to talk about concrete values. In particular, `\old` may be used in a sound way and it does always yield the same value for a fixed M.

[22] Since representation clauses are not allowed to use `\old`, `\result`, etc., it is sufficient to consider valuation in a single state s'. I.e. all other components of the logical state are defined arbitrarily. Representation clauses may still refer to other model fields, but they behave as if they were regular fields in that they are valuated through the Heap.

This approach also allows the use of any JML expression within representation clauses. This is in contrast to [LM06], which explicitly disallows any method invocations. It is also possible to use side-effects of weakly pure methods in representation clauses, which are evaluated at every occurrence of the method field reference.

```
public class Sqrt {
    private int x;
    /*@ public model int z;
      @ represents z \such_that x >= 0 ==>
      @     (x-1 < z*z && z*z <= x);
      @*/

    /*@ public normal_behavior
      @     requires 0 <= x
      @         && x <= Integer.MAX_VALUE/4;
      @     ensures  z == 2 * \old(z);
      @*/
    public void times4 () {
        x = x * 4;
    }
}
```

Figure 3.2: Example for a model field with a concrete value.

Example 3.6. Consider the definition of class `Sqrt` in Fig. 3.2. It declares a model field z which holds a square root of a positive integer x (rounded towards zero). For given x, there are only two possible values for z – positive or negative – or just one if x valuates to zero. Nothing is said about the case in which x is negative.

Let $o \in V_{\texttt{Sqrt}}$ be an instance of this class. Let $s_j = (h_j, \sigma, \chi)$ with $j \in V_{\texttt{int}}$ be a family of system states which coincide on everything except for $h_j(o, x) = j$ for every $j \in V_{\texttt{int}}$. Then, $\mathcal{M}$ can be partitioned in two subsets $\mathcal{M}^+$ and $\mathcal{M}^-$, such that for every $M^\pm \in \mathcal{M}^\pm$ (for $\pm \in \{+, -\}$) it is

$$M^\pm(s_j, o, z) = \pm \left\lfloor \sqrt{val_{s_j}(x)} \right\rfloor = \pm \left\lfloor \sqrt{j} \right\rfloor \text{ for } j \in [0, 2^{31} - 1]$$

(The values for $j < 0$ are of no interest to us.) Let us now fixate a valuator function $M \in \mathcal{M}^+$.

Finally, we will regard `times4` and its postcondition. Let s_k be its prestate with $k \in [0, 2^{29} - 1]$ (this is what the precondition says). Then, since we excluded a possible overflow, we can assume a post-state s_{4k} (without further looking at details). Thus, the postcondition is evaluated in a logical state $\Sigma = (s_k, s_{4k}, L_a, L_w, M, \rho)$, where L_a, L_w, ρ are unimportant to our considerations. It is $val_\Sigma(\texttt{z == 2*\textbackslash old(z)}) = true$ if and only if $val_\Sigma(\texttt{z}) = val_\Sigma(\texttt{2*\textbackslash old(z)})$.

It is $val_\Sigma(\texttt{z}) = M(s_{4k}, o, z) = \left\lfloor\sqrt{4k}\right\rfloor = 2\left\lfloor\sqrt{k}\right\rfloor$ and

$$\begin{aligned} val_\Sigma(\texttt{2*\textbackslash old(z)}) &= 2 \cdot_{\texttt{int}} val_\Sigma(\texttt{\textbackslash old(z)}) \\ &= 2 \cdot_{\texttt{int}} val_{(s_k, s_k, L_a, L_w, M, \rho)}(\texttt{z}) \\ &= 2 \cdot_{\texttt{int}} M(s_k, o, z) \\ &= 2 \cdot_{\texttt{int}} \left\lfloor\sqrt{k}\right\rfloor = 2\left\lfloor\sqrt{k}\right\rfloor \end{aligned}$$

As it can be seen, in both pre- and post-state the same valuator function M is used to valuate the model field. Thus, we can conclude that the postcondition holds.

The only question left open so far is: What if there is no model field valuator to satisfy the representation clauses?[23] In the following, we will always formulate semantics of specifications under the assumption that there exists at least one. Therefore, we defined the set $\mathfrak{M}$ in a way that it always contains at least one element, the empty set. In this case, there is no value for any model and we insert underspecified values. Every reference to a model field will be marked as not well-formed. However, expressions which do not depend on the values of *any* model field are still satisfiable. The following example has no model, but the invariant is still tautological:

```
public class NoModel {
    //@ model int z;
    //@ represents z \such_that false;
    //@ invariant true || z == 42;
}
```

3.1.6 Axioms

For a final definition of validity we have still to consider that JML possesses the notion of *axioms*, which are assumed "whenever such an assumption is needed" [LPC+08, Sect. 8.6]. This definition is not very precise. With an intuitive meaning of axioms in mind, we will introduce an extended definition of validity which builds on the previous one.

[23]As a reminder, valuator functions are defined to have to satisfy *all* representation clauses at the same time.

The reference manual does not say anything additional about syntax and semantics of axioms. We assume an axiom to be a JML-E_0 expression, i.e. it must not contain one of the `\old`, `\result` or frame expressions. We further assume that axioms are always non-static and apply on created non-null (as this is the default in JML) instances of the type where it is defined. In contrast to most other specification features, the privacy of axioms cannot be modified.[24] From this, we also conclude that axioms are not meant to be inherited.

Definition 3.7 (Validity of expressions, 3rd approach)**.** Let ϕ be a normalized boolean expression and Σ a logical state. ϕ is valid in Σ, written $\Sigma \vDash \phi$, *iff*: $\Sigma \vDash \phi$, *or* there is an axiom $\mathcal{X}$ defined in type T and an instance $o \in V_T^0$ such that

$$\Sigma' \nvDash \mathcal{X}$$

where Σ' coincides with Σ except for $val_{\Sigma'}(\texttt{this}) = o$
and $val_{\Sigma'}(\texttt{this.\textbackslash created}) = \mathit{true}$.

In praxis, the use of axioms is very similar to those of invariants. However, axioms are assumed in every state, but never asserted.

3.2 Data groups

As we have discussed before on multiple occasions, JML allows the compilation of several locations into *data groups*. They are especially useful when JML's frame conditions (e.g. `\assignable`, see Sect. 4.2) are used. Originally, data groups were introduced to cope with method overrides which change more locations than the overridden method [Lei98]. (See also the example in Fig. 3.3.) The set of locations in a data group can be seen as a refinement from a "more crude" location which denotes the data group. In addition, data groups also allow model fields (see Sect. 3.1.5) to be used within frame conditions. As it has been discussed above, model fields do not have (concrete) locations where they are stored. We will use data groups of abstract locations to refer to the concrete locations on which they depend.

As a foundation, every concrete field in JML defines a data group with the same name and with itself as the only element. Every model field also defines a data group which is intended to include every concrete location on which it (transitively) depends. This is however not required by the definition of the data group.

[24]This is actually mentioned in the reference manual.

```
1  public class Square {
2      protected double width;
3      //@ public model double area;
4      //@ private represents area = width * width;

6      /*@ public normal_behavior
7        @      ensures    \result == area;
8        @      accessible width;
9        @*/
10     public double getArea () {
11         return Math.pow(width, 2);
12     }
13 }

15 public class Rectangle extends Square {
16     protected double length;
17     //@ private represents area = width * length;
18     //@ maps length \into width;

20     // (specification inherited from above)
21     public double getArea () {
22         return width * length;
23     }
24 }
```

Figure 3.3: Example of inherited method specifications. `Rectangle` inherits a specification which would be unsatisfiable without the data group mapping in Line 18 since `length` is also accessed but not listed in the `accessible` clause. With this clause, `length` is included in the set of accessible locations. The model field `area` is used to show the abstract property which is used in both classes.

Finally, one can define data groups to be included in each other. JML defines two variants: *Static inclusion* is written immediately after the definition of the field to include using the keyword `in`. *Dynamic mapping* is the more general form, though it is also written right after a field declaration. It declares a more general *location expression* to be included in a list of data groups using the keyword-pair `maps`-`\into`. An example is given in Fig. 3.3. The static kind can easily be desugared to the dynamic one, so we only have to consider mappings of the dynamic kind:

```
int z; //@ in x;   ===>   int z; //@ maps this.z \into x;
```

A *location expression*[25] is constructed in the same way as a reference expression, i.e. an expression which refers to a field. Location expressions however refer to (abstract) *locations* as the name suggests. Therefore, they are evaluated through a different function $loc : \mathcal{E} \to 2^{\mathcal{L}'}$ which is given below. In addition to "classical" references, location expressions may use wild-cards, such as *array*`[*]` to refer to all fields resp. array elements, as well as the special keywords `\nothing` and `\everything`.

Definition 3.8 (Location expressions)**.** Let s be a system state.

- $loc_s(r.x) = \{(val_s(r), x)\}$ for an expression r of reference type and x a (possibly model) field identifier
- $loc_s(C.x) = \{(o, x) \mid o \in V_C^0\}$ for a static (possibly model) field identifier x in class C
- $loc_s(r.*) = \{(val_s(r), \iota) \mid \iota \in \mathbb{I}\} \cap \mathcal{L}'$
- $loc_s(C.*) = \{(o, \iota) \mid o \in V_C^0, \iota \in \mathbb{I}\} \cap \mathcal{L}'$
- $loc_s(a\texttt{[}n\texttt{]}) = \{(val_s(a), val_{\omega(a)(s)}(n))\}$ for an expression a of array type and n expression of an integer type
- $loc_s(a\texttt{[}n..m\texttt{]}) = \{(val_s(a), k) \mid val_{\omega(a)(s)}(n) \leq k \leq val_{\omega(n)(\omega(a)(s))}(m)\}$ for an expression a of array type and n, m expressions of integer types
- $loc_s(a\texttt{[*]}) = loc_s(a\texttt{[}0 .. a.\texttt{length} - 1\texttt{]})$
- $loc_s(e_1, e_2, \ldots, e_n) = \bigcup_{i=1}^{n} loc_{\vec{\omega}_{i-1}(s)}(e_i)$ with $\vec{\omega}_i := \omega(e_i) \circ \omega(e_{i-1}) \circ \ldots \circ \omega(e_1)$ and $\vec{\omega}_0 := id$ for a list of references
- $loc(\texttt{\nothing}) = \emptyset$

[25]This is called "Store Ref" in the JML reference manual [LPC+08, Sect. 11.7].

- $loc(\texttt{\textbackslash everything}) = \mathcal{L}'$

In the declaration of data groups not all these expressions may be used. The left-hand side of a maps-into-clause may only contain concrete references and use simple references, such that $x.y$, $x.*$, $x\texttt{[}n\texttt{]}$, $x\texttt{[}n..m\texttt{]}$, $x\texttt{[*]}$, $x\texttt{[}n_1\texttt{][}n_2\texttt{]},\ldots$ where x is meant to refer to the field which has just been declared.

Definition 3.9 (Data group)**.** Let s be a system state. The least function $\mathcal{D} : \mathcal{L}' \to 2^{\mathcal{L}}$ satisfying the following assigns a location to its data group.

- Base case: $\ell \in \mathcal{D}(\ell)$ for every concrete location $\ell \in \mathcal{L}$
- Data group mapping: For every instance o of class C and the following (normalized) clause in C `maps` x `\into` $r_1.z_1\texttt{,}\ldots\texttt{,}r_n.z_n\texttt{;}$ (where x is a location expression and $r_i.z_i$ are references to data groups[26]) it is

$$\bigcup_{\ell \in loc_{s'}(x)} \mathcal{D}(\ell) \subseteq \bigcap_{i=1}^{n} \mathcal{D}((val_{s'}(r_i), z_i))$$

  where s' coincides with s except for $val_{s'}(\texttt{this}) = o$.

While this formal definition is a very straight forward translation of the informal description, it is not clear to see that such a function does always exist. But, at least the set of all locations $\mathcal{L}$ satisfies both requirements.

3.3 JML expressions

This section discusses the details of valuation of some representative JML expressions. In particular, in some places, in which the JML reference manual [LPC⁺08] is not clear enough, we will comment on our conclusions in depth. To give a more simple view, in most cases we omitted the states in which subexpressions are evaluated.

The structure of this section is mirrored by Appendix A, which contains a comprehensive enumeration of definitions for all covered expressions.[27]

[26]The requirement for r_i be concrete has been made in [Lei98].

[27]Refer to Appendix B for JML expressions which are not covered.

3.3.1 Boolean expressions

The values of the logical operators `!`, `&&`, `&`, `||`, `|` are the exact same as in Java. JML's addition to these compromises of implications in both directions, `==>` and `<==`, as well as equivalence `<==>` and antivalence `<!=>`. All of them are short-circuit operators. This means both well-definition and side-effects depend on the value of the first operand. Consider for example the forward-implication:

- $val_s(a \texttt{ ==> } b) = true$ **iff** $val_s(a) = false$ or $val_{\omega(a)(s)}(b) = true$
- $wd_s(a \texttt{ ==> } b)$ **iff** $wd_s(a)$ holds, **and** $val_s(a) = false$ or $wd_{\omega(a)(s)}(b)$ holds
- $\omega(a \texttt{ ==> } b)(s) = \begin{cases} \omega(a)(s) & val_s(a) = false \\ \omega(b)(\omega(a)(s)) & \text{otherwise} \end{cases}$

Equality predicates

Equality and inequality predicates are the exact same as in Java. It should be noted however, that we assume them to be untyped on the semantical level:

- $val_s(a \texttt{ == } b) = true$ **iff** $val_s(a) = val_{\omega(a)(s)}(b)$ and $val_s(a) \neq NaN$
- $wd_s(a \texttt{ == } b)$ **iff** $wd_s(a)$ and $wd_{\omega(a)(s)}(b)$
- $\omega(a \texttt{ == } b) = \omega(b) \circ \omega(a)$

The syntax of JML already requires equality to be typed and thus only comparisons of compatible types are well-formed. Numerical expressions on the other hand may always be compared with each other; in this case a numerical promotion procedure takes place.[28] NaN is incomparable with itself according to the IEEE 754 standard [Kah87].

3.3.2 Numerical expressions

Unlike most other specification languages, JML semantics by default rely on Java integers and floating point numbers, rather than mathematical integers and real numbers. This means that overflows (integers) and missing precision (floats) also occur in specifications. Although this approach is not unsound, it might lead to some unexpected results, e.g. `Integer.MIN_VALUE * 2 == 0`.

[28] This may lead to some unexpected results. E.g. $val(16777217 \texttt{ == } 16777216.0) = true$ since promotion from `int` to `float` comes with a loss of precision.

Support for mathematical integers and real numbers has been added meanwhile [Cha03], introducing the specification-only types `\bigint` and `\real`. However, numerical expressions are still evaluated according to Java rules. It has been proposed in [Cha04] to implement different *math modes*:

1. *Java math*, which uses Java rules as mentioned above,
2. *Bigint math*, in which every numerical expression is implicitly converted to the respective mathematical entity and
3. *Safe math*, in which overflows throw exceptions.

In this paper, we will stick to the "classical JML", i.e. Java math. It would not be much work given semantics for the other cases, but it would not be very informative. Also, we would like to avoid switching between the different modes (see Appendix B).

Numerical expressions behave almost as in Java, i.e. with overflow semantics for integral types[29] and floating point types behaving according to [Kah87]. We added semantics for `\bigint` and `\real` types. Since nothing is said in the reference manual, we assume the same operations as for floating point types. This example shows division for `\real`:

- $val(x/y) = \begin{cases} \text{arbitrary} & val(y) = 0 \\ val(x)/val(y) & \text{otherwise} \end{cases}$
- $wd(x/y)$ **iff** $wd(x)$ and $wd(y)$ **and** $val(y) \neq 0$

Division by zero has to treated separately however. Although $wd(x/y)$ is not satisfied in this case[30], we still have to assign a legal, but underspecified value in order for *val* to be a total function. Since division by zero always leads to falsity on the top-level, e.g. `n/0 == n/0` is not valid, but the concrete value is in fact unimportant.

This applies to integral types, too. But not to floating point types, since every division by zero results in NaN which is a built-in element for undefined numerical values in IEEE 754. E.g. `n/0.0 != n/0.0` is a valid expression (and a tautology).

[29]Consider for instance multiplication in type `int`:

$$val(\texttt{n * m}) = (val(\texttt{n}) \cdot val(\texttt{m}) + 2^{31}) \mod 2^{32} - 2^{31}$$

[30]Division by zero would cause an `ArithmeticException` in Java.

3.3.3 Type expressions

In JML, there can be expressions of type `\TYPE`, which is meant to represent the set of Java types. These type expressions can be used in comparisons, i.e. equality predicates, the subtype predicate or the `instanceof` predicate.

`\typeof` returns the dynamic type of an expression of reference type. Since null has no dynamic type, in this case the expression is not well-defined. In all other case, there is an unambiguous value set which contains the element.

When applied to expressions of a primitive type, `\typeof` does return an according wrapper type, e.g. `java.lang.Integer` for expressions of type `\int`. According to the reference manual this is always “the most-specific” dynamic type of the expression’s value. However, for numerical expressions it might be ambiguous which type to chose since value sets are neither exclusive nor subsets of each other, e.g. $\emptyset \subsetneq V_{\mathtt{int}} \cap V_{\mathtt{float}} \subsetneq V_{\mathtt{int}}$. We assume that integral types are “more specific” than floating-point numbers in this context. This leads to the following definition:

$$val(\mathtt{\backslash typeof}(e)) = \begin{cases} \mathtt{Boolean} & val(e) \in V_{\mathtt{bool}} \\ \mathtt{Character} & val(e) \in V_{\mathtt{char}} \\ \mathtt{Byte} & val(e) \in V_{\mathtt{byte}} \\ \mathtt{Short} & val(e) \in V_{\mathtt{short}} \setminus V_{\mathtt{byte}} \\ \mathtt{Integer} & val(e) \in V_{\mathtt{int}} \setminus V_{\mathtt{short}} \\ \mathtt{Long} & val(e) \in V_{\mathtt{long}} \setminus V_{\mathtt{int}} \\ \mathtt{Float} & val(e) \in V_{\mathtt{float}} \setminus V_{\mathtt{long}} \\ \mathtt{Double} & val(e) \in V_{\mathtt{double}} \setminus V_{\mathtt{float}} \\ \tilde{T} & val(e) \in V^0_{\tilde{T}} \text{ and } \tilde{T} \sqsubseteq \mathtt{Object} \\ \text{arbitrary} & \text{otherwise} \end{cases}$$

The last case applies when e denotes an expression of a type other than a Java type, e.g. `\bigint` or `\TYPE`. According to the reference manual, JML syntax does not outlaw them, but `\TYPE` is defined only to contain Java types.

$$wd(\mathtt{\backslash typeof}(e)) \textbf{ iff } val(e) \in \bigcup_{T \in V_{\mathtt{\backslash TYPE}}} V^0_T$$

`\elemtype` returns the type of elements for a given (Java) array type represented by an expression t of type `\TYPE`. If the given type is not array type, then the elements are per definition of “type null”. Since non-Java array types like `\bigint[]` are excluded from the value set of `\TYPE`, the result type is a Java type, too. Therefore, this expression is well-defined if and only if the `\TYPE` expression is well-defined.

$$val(\texttt{\textbackslash elemtype}(t)) = \begin{cases} T' & val(t) = T'\texttt{[]} \\ \text{null} & \text{otherwise} \end{cases}$$

$$wd(\texttt{\textbackslash elemtype}(t)) \textbf{ iff } wd(t)$$

3.3.4 Type cast

Casting an expression of reference type to another reference type does not change its semantics:

$$val(\texttt{(}T\texttt{)}\ a) = val(a)$$

As Java would throw a `ClassCastException`, this expression is not well-defined if the casted element a is not an instance of the type T:

$$wd(\texttt{(}T\texttt{)}\ a) \qquad \textbf{iff} \qquad val(\texttt{(}T\texttt{)}\ a) \in V_T$$

Note, that instances of `\bigint[]`, `\real[]`, `\TYPE[]`, etc. are not compatible to `Object` and cannot be used in a well-defined cast expression.

In the case that both types are numerical, there is a special conversion procedure in Java [GJSB00, Chapter 5] which ensures that there is always an element in the domain of the target type. E.g. $val(\texttt{(int) NaN}) = 0$. We included `\bigint` and `\real` in an extended conversion procedure for JML numerical types in the obvious way.[31] In the opposite direction, values are usually rounded or set to infinite. For instance the conversion from `\real` to `double`:

$$val(\texttt{(double)}a) = \begin{cases} +\infty & a > \max \mathbb{Q}_{\texttt{real}} \\ -\infty & a < \min \mathbb{Q}_{\texttt{real}} \\ \text{rtn}_{V_{\texttt{real}}} & \text{otherwise} \end{cases}$$

However, we decided that the conversion of NaN or infinite values should also lead to the throw of an exception. (After all, NaN stands for "not a number".) Therefore, it is $val(\texttt{(\textbackslash real) NaN}) = NaN$, but since $NaN \notin \mathbb{R}$ the expression is not well-defined.

3.3.5 Reference expressions

The values of concrete references are determined from the system state $s = (h, \sigma, \chi)$. Local variables (including parameters and caught exceptions) are

[31]The reference manual is not clear on whether such a conversion procedure exists in JML. If not, there would be not way of comparing Java primitive numerical types with `\bigint` or `\real`.

valuated through the stack σ. The value of `this` is determined from the top element of the call stack χ. Fields and array elements are valuated through the heap h:

- $val_s(v) = \begin{cases} \sigma(v) & v \in \mathrm{dom}(\sigma) \\ \text{arbitrary} & \text{otherwise} \end{cases}$ for a local variable v
- $val_s(\texttt{this}) = o$ where $(\mu, o) = \mathrm{top}(\chi)$ for some method μ
- $val_s(r.x) = \hat{h}(val_s(r), x)$ for an expression r of reference type and x a non-static non-model field identifier where $\omega(r)(s) = (\hat{h}, \dots)$
- $val_s(T.x) = h(o, x)$ for a static non-model field x defined in type T with $o \in V_T$
- $val_s(a\texttt{[}n\texttt{]}) = \check{h}(val_s(a), val_{\omega(a)(s)}(n))$ for an expression a of array type and n expression of an integer type with $\omega(n)(\omega(a)(s)) = (\check{h}, \dots)$

Identifiers which are not local variables, i.e. they do not belong to the current method frame, are outside the domain of the Stack σ. Their values are underspecified. Therefore, expressions including local variables are only well-defined if the variable is on the Stack:

$$wd_s(v) \textbf{ iff } v \in \mathrm{dom}(\sigma)$$

For a reference chain, the value is determined recursively, e.g. $val(a.b.c) = h(h(val(a), b), c)$. For a normalized expression, such a chain always terminates in either a local variable or `this`. Note that since each expression may contain (weakly pure) side-effects, it is not necessarily the same heap function applied on each reference, e.g. val(`new Integer(5).value`) $= 5$.

Non-static references are well-defined if and only if the receiver is evaluated without exceptions, it is not null (Java would throw a `NullPointerException` in that case) and it is created:

$$wd_s(r.x) \textbf{ iff } wd_s(r), val_s(r) \neq \mathit{null} \text{ and } val_s(r.\texttt{\textbackslash created}) = \mathit{true}$$

One might wonder why well-definition requires the receiver object to be created since in Java there would be no reference with non-created receivers. In JML however, there are possibilities to refer to objects which are not created, e.g. through quantification. For instance, the following expression does *not* valuate to true since there definitely are objects of type `Integer` which are not created (in particular infinitely many):

```
(\forall Integer z; true;
    z.value() <= Integer.MAX_VALUE)
```

Static references on the other hand are always well-defined since there is no receiver to be determined from an expression.

Well-definition of array expressions also requires the index to be inside bounds, since otherwise Java would throw an `ArrayIndexOutOfBoundsException`: $wd_s(a\texttt{[}n\texttt{]})$ **iff** $wd_s(a)$, $val_s(a) \neq null$, $val_s(a.\backslash\texttt{created}) = true$, $wd_{\omega(a)(s)}(n)$ and $0 \leq val_{\omega(a)(s)}(n) < val_s(a.\texttt{length})$

Model fields

In contrast to concrete fields, model fields do not denote locations (see Sect. 2.3). As explained in Sect. 3.1.5, we use a special evaluation function $M : \mathcal{S} \times \mathcal{U} \times \mathbb{I} \to \mathcal{U}$, called model field valuator, to assign a concrete value to the abstract location of a model field depending on the current system state of evaluation. The model field valuator is part of a logical state $\Sigma = (s_0, s_1, L_a, L_w, \rho, M)$, thus evaluation depends on Σ:

- $val_\Sigma(r.x) = \begin{cases} M(\omega(r)(s_1), val_{s_1}(r), x) & M \neq \emptyset \\ \text{arbitrary} & M = \emptyset \end{cases}$ for an expression r of reference type and x a non-static model field identifier
- $val_\Sigma(T.x) = \begin{cases} M(s_1, o, x) & M \neq \emptyset \\ \text{arbitrary} & M = \emptyset \end{cases}$ for a static model field x defined in type T with $o \in V_T$

In the case that a representation clause is unsatisfiable, there is no legal model field valuator function and $M = \emptyset$. Any expression referring to model fields in this case is not well-defined. This would probably correspond to a `NoSuchFieldException` in Java. The full definition of well-definition for a non-static model field thus is the following:

$$wd_\Sigma(r.x) \textbf{ iff } wd_\Sigma(r), val_\Sigma(r.\backslash\texttt{created}) = true, val_\Sigma(r) \neq \text{null and } M \neq \emptyset$$

Since representation clauses for model fields may have side-effects, we have to take those into account on every evaluation of a model field reference. Consider for instance the following declaration, in which a new object is created[32] every time `h` is referred to.

```
private static char[] hw = {'H','e','l','l','o',
        '␣','W','o','r','l','d'};
/*@ public model String h;
  @ represents h \such_that h ==  new String (hw);
  @*/
```

[32]Creation of `String` objects using double-quotes, e.g. `"Hello World"`, is not a legal expression in JML. There is no string concatenation operator `+` neither.

Thus for a model field $r.x$ with $(val(r), x, \psi) \in \mathfrak{M}$ it is $\omega(r.x) = \omega(\psi) \circ \omega(r)$.

3.3.6 Pure method invocation

As it has been explained in Sect. 3.1.3, pure methods may be used in expressions almost like mathematical functions. In particular, every reachable system state qualifies as pre-state for a pure method since it has a trivially true precondition.

We will assume a pure non-static method being invoked, the static case behaves very similar. Let $s = (h, \sigma, \chi)$ be a system state, a an expression of reference type, m a method identifier referring to a (non-void) non-static method and $e_1, \dots, e_n$ expressions of respective types $T_1, \dots, T_n$.

Consider now the call in a state s through the expression $a.m(e_1, \dots, e_n)$. First of all, the method to be invoked has to be determined.[33] Let

$$\mathrm{msp}(m, \langle T_1, \dots T_n \rangle, val_s(a)) = (C, m, \pi, \langle (T_1, \iota_1), \dots, (T_n, \iota_n) \rangle, T_R) =: \mu$$

As a reminder, the tuple μ is made up of a class C, where it is declared, an identifier m, a method body π, a (possibly empty) sequence of pairs (T_j, ι_j) of types and identifiers for parameters and a return type T_R.

Secondly, the values of passed parameters including the receiver have to be evaluated. Since they might have side-effects, we define state transition functions $\vec{\omega}_j$ for $0 \leq j \leq n$:

$$\vec{\omega}_j := \omega(e_j) \circ \omega(e_{j-1}) \circ \dots \circ \omega(e_1) \circ \omega(a)$$

Next, we construct the pre-state $\tilde{s}$ of the invocation by assigning the (explicitly) passed values to the parameter identifiers ι_j via updates on the stack. The method μ with receiver $val(a)$ is pushed onto the call stack:

$$\tilde{s} := \vec{\omega}_n((h, \sigma^{\{\iota_j \mapsto val_{\vec{\omega}_{j-1}(s)}(e_j) | 1 \leq j \leq n\}}, \mathrm{push}(\chi, (\mu, val_s(a)))))$$

Finally, we can make use of the black box function. Since pure methods are required to terminate, there is always a post-state, which is reachable by definition.

$$\mathcal{J}(\tilde{s}, \pi) = (\langle s_0, \dots, s_k \rangle, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$$

λ and α are of no interest to us. We first concentrate on the return value $\rho \in V_{T_R}$. This will be the value of the above expression:

$$val_s(a.m(e_1, \dots, e_n)) = \rho$$

[33]For a static method, this would not be needed since there is no overriding.

For this expression to be well-defined, the receiver expression a and all parameter expressions have to be well-defined. Also the pre-state must be reachable, i.e. a must be created and not refer to the null object.[34] And finally, pure methods are still allowed to terminate with an exception thrown. So, it also has to be $\Omega = \text{null}$.[35]

$$wd_s(a.m(e_1, \dots, e_n)) \textbf{ iff } \begin{cases} \bullet\ wd_s(a) \\ \bullet\ val_s(a) \neq \text{null} \\ \bullet\ val_s(a.\texttt{\textbackslash created}) = true \\ \bullet\ wd_{\vec{\omega}_{i-1}}(e_i) \\ \bullet\ \Omega = \text{null} \end{cases}$$

At last, to determine the value of ω, we examine the post-state $s' = (h', \sigma', \chi')$. It does not necessarily equal the pre-state. Firstly, the stack functions differ on the values of local variables including parameters. This is unimportant however; since these are local to the method, they have to be reset to the previous values. The significant change might have occurred to the heap function if the execution of μ included the (pure) creation of new objects. The call stack is the same as in the constructed pre-state. Thus, the successor state to the original system state s coincides on heap with s' and on stack with s.

$$\omega(a.m(e_1, \dots, e_n))(s) = (h', \sigma, \chi)$$

Clearly, this state is reachable, too.

Pure constructors

The invocation of pure constructors is very similar to that of pure methods. The difference is, that black box execution of the constructor body only corresponds to the initialization phase (see Sect. 2.4.1). The pre-state to construct has already to include the creation and preparation of the new object. We assume ρ to be a "fresh" domain element of the respective type and to be fetched with the same mechanism as the black box would do. The pre-state heap $\tilde{h}$ then coincides with the original h except for $\tilde{h}(\rho, \texttt{\textbackslash created}) = true$ and every non-static field x holds its default value: $\tilde{h}(\rho, x) = d(T(x))$. ρ then serves as the receiver for the constructor invocation. The remaining procedure is analogous to above.

[34]This is analogous to well-definition of reference expressions.

[35]In all other cases, ρ is underspecified according to our definition of the black box (Sect. 2.4).

3.3.7 Array creation

Arrays are created very similar to Java.[36] The only difference is that JML requires using the `new` keyword, while in Java arrays may be created just by initializers. Creation of multi-dimensional arrays however includes the creation of "lower-dimensional" arrays, e.g. `new int[n][m]` includes the creation of m new arrays of type `int[]` with length n. Using array initializers within this "nested creation" is allowed in JML. For instance `new int[][] { {0}, {0,1}, {0,1,2} }` is a legal JML expression. Therefore, we included the definition of initializers in this expression reference, even though they must not be declared solely.

Array initializer

An array initializer consists of a list of n expressions $e_0, \dots, e_{n-1}$ of respective types $T_0, \dots, T_{n-1}$. These types are required to be comparable with each other, i.e. there is a least common super-type T to which all of them may be cast without raising an exception.[37]

Since the expressions within the initializer might have side-effects, we need to define $\vec{\omega}_j := \omega(e_j) \circ \omega(e_{j-1}) \circ \cdots \circ \omega(e_0)$ with $\vec{\omega}_{-1} := id$ and $\vec{\omega}_{n-1}(s) = s'$.

The value of the initializer expression then is just a fresh domain element $a \in V^0_{T\texttt{[]}}$:

$$val_s(\texttt{\{}e_0, \dots, e_{n-1}\texttt{\}}) = a$$

Well-definition depends on well-definition of the sub-expressions:

$$wd_s(\texttt{\{}e_0, \dots, e_{n-1}\texttt{\}}) \textbf{ iff } wd_{\vec{\omega}_{j-1}(s)}(e_j) \text{ for every } j \in [0, n-1]$$

The state of evaluation is altered in the way that a is created has a length of n and its components evaluate to the values passed by the initializer:

- $\omega(\texttt{\{}e_0, \dots, e_{n-1}\texttt{\}})(s) = s''$ where $s'' := (h'', \sigma'', \chi'')$ coincides with s' except for
  - $h''(a, \texttt{\textbackslash created}) = true$

[36] The JML reference manual only mentions that there are array creation expressions, but does not provide semantics. To assume no difference to Java seems most reliable in this case.

[37] Due to multiple inheritance via interface implementation, it is ambiguous to chose a common super-type. In our approach, this already decides the dynamic type of the array and possibly turns `\typeof` undecidable. For example: `\typeof(new Object[] {new Short(23), new Long(42)})` could both yield `Number[]` or `Comparable[]`. For now, we just assume that there is an appropriate type which is chosen.

- $h''(a, \texttt{length}) = n$
- $h''(a, i) = val_{\vec{\omega}_{i-1}}(e_i)$ for every $i \in [0, n-1]$

New array declaration

For an array declaration using the keyword `new`, we only consider the general case in which both all dimensions and complementing initializers (of the correct length) are given. For any other case refer to the Java language specification [GJSB00, Sect. 15.10]. We consider the creation of a k-dimensional array over a type T. The dimension expressions $n_1, \ldots, n_k$ are expressions of type `int` and $\underline{init}$ is an array initializer. The dimension expressions might have side-effects, so let $\vec{\omega}_j := \omega(n_j) \circ \omega(n_{j-1}) \circ \cdots \circ \omega(n_1)$.

The value of the whole expression mostly depends on the initializer, though it is evaluated after the dimension expressions:

$$val_s(\texttt{new}\ T\texttt{[}n_1\texttt{]}\,\texttt{[}n_2\texttt{]} \cdots \texttt{[}n_k\texttt{]}\,\underline{init}) = val_{\vec{\omega}_k(s)}(\underline{init})$$

Negative dimensions would cause Java to throw a `NegativeArraySizeException`. And passing an incompatible initializer to the `new` expression results in an `ArrayStoreException`. Thus, both has to be satisfied in order for the expression to be well-defined.

- $wd_s(\texttt{new}\ T\texttt{[}n_1\texttt{]}\,\texttt{[}n_2\texttt{]} \cdots \texttt{[}n_k\texttt{]}\,\underline{init})$ **iff**
  - for every $i \in [1, k] : val_{\vec{\omega}_{i-1}(s)}(n_i) \geq 0$,
  - $wd_{\vec{\omega}_k(s)}(\underline{init})$ and
  - $val_{\vec{\omega}_k(s)}(\underline{init}) \in V_{T\texttt{[]}\ldots\texttt{[]}}$

3.3.8 Referring to old values

The `\old` expression is used in postconditions and history constraints to refer to values of the pre-state. Since our valuation function is always equipped with two states of valuation, valuation of `\old` causes the "current state" to be replaced by the pre-state:

$$val_{(s_0,s_1)}(\texttt{\textbackslash old}(expr)) = val_{(s_0,s_0)}(expr)$$

Nested uses of `\old` are ignored thereby.

Unlike OCL's @*pre* operator [OCL05] which can only be applied to references, `\old` may enclose any expression. However, JML's syntax allows `\old` only to be applied to prefixes of references since suffixes are no expressions on their own. E.g. $a.b.$`\old`(c) and $a.$`\old`$(b).c$ are not well-formed expressions, while `\old`$(a).b.c$, `\old`$(a.b).c$ and `\old`$(a.b.c)$ are valid. The expression `\old`$(a.b.c)$ then is equivalent to the OCL expression $a@pre.b@pre.c@pre$. As a consequence, certain (meaningful) expressions are not allowed and objects of the pre-state are not referable. An example can be seen in Fig. 3.4.

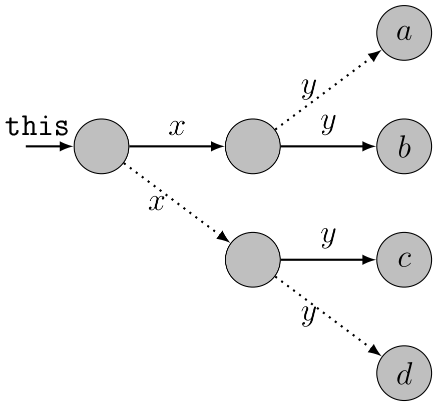

Figure 3.4: Example showing the meaning of `\old`. (Semantical) objects are represented by nodes, while references are shown as arrows. Solid lines represent references in the pre-state, dotted lines in the post-state. We can refer to a as `\old`$(x).y$, to b as `\old`$(x.y)$ and to d as $x.y$, but there is no way to refer to c.

In contrast to [DM06], we consider all side-effects of the enclosed expression to be without effect on the outside, i.e. $\omega($`\old`$(expr)) = id$ for any expression. The reason for that is not to make the successor function complicated in that it describes transition for *both* states. This decision does only affect a few pathological cases, namely the creation of new objects through weakly pure methods *within* the `\old` clause. It seems questionable which sense an expression like the following should make.

```
        \old( new Object () ) == o
```

It looks a little bit like traveling back in time and changing the future... However, any expression enclosed in the `\old` clause may create new instances and thus alter the state of evaluation *within* the clause. E.g. the following expression is valid.

$$val(\texttt{\textbackslash old(new Integer(23).intValue() == 23)}) = true$$

3.3.9 Type predicates

The subtype predicate `<:` yields true if the first type (or rather expression of type `\TYPE`) is a subtype of the second. It is defined to be reflexive on the domain of `\TYPE`, so this applies to null and void, too, even though they do not denote types.

- $val(t_1 \texttt{ <: } t_2) = true$ **iff** $val(t_1) \sqsubseteq val(t_2)$ or $val(t_1) = val(t_2) \in \{\text{null}, \texttt{void}\}$
- $wd(t_1 \texttt{ <: } t_2)$ **iff** $wd(t_1)$ and $wd(t_2)$

The same considerations apply to the instance-of predicate which yields true if the referenced element e is of static type t_2. This is as in Java, with the exception that the right-hand side may be an expression of type `\TYPE`. Except from null, there is always an unambiguous value set to contain e.

- $val(e \texttt{ instanceof } t_2) = true$ **iff** $T' \sqsubseteq val(t_2)$ with $val(e) \in V^0_{T'}$
- $wd(e \texttt{ instanceof } t_2)$ **iff** $wd(e)$, $wd(t_2)$, and $val(e) \neq null$
- $\omega(e \texttt{ instanceof } t_2) = \omega(t_2) \circ \omega(e)$

Null references are allowed to appear on the right-hand side, but always yield $false$ since null is not a type.

3.3.10 Expressing frame properties

Beneath using frame clauses in method specifications, there are boolean predicates to express frame properties, such as accessed or assigned locations, in postconditions and history constraints.

In JML, these properties are strict. If a location has the same value in both pre- and post-state, it does not necessarily mean it hasn't been assigned. Therefore, JML possesses different predicates for those different meanings: `\not_modified` yields whether the values equal, while `\not_assigned` yields whether the locations were assigned at all. The two other predicates, `\only_accessed` and `\only_assigned` on the other hand list those locations which may be accessed resp. assigned. In that sense they are similar to the frame properties (`accessible` and `assignable` clauses) in method specifications (see 4.2.1).

Each frame predicate takes a list of references to (possibly abstract) locations which is being evaluated using the *loc* function of Sect. 3.2 and in turn the corresponding data group (which contains only concrete locations) is compared to the set of locations which were affected (i.e. resp. accessed or assigned) during execution (those are the sets L_a resp. L_w as part of the logical state passed with the valuation function).

Let us consider the `\only_assigned` predicate for instance:

$$val_{(s_0,s_1,L_a,L_w,\rho,M)}(\texttt{\textbackslash only\_assigned}(\Lambda)) = true \quad \textbf{iff} \quad L_w \setminus \bigcup_{\ell \in L_\Lambda} \mathcal{D}(\ell) \subseteq L^*$$

It means that only the locations of the data group of list Λ may be assigned. In this definition, all locations which were "not present" in the pre-state, i.e. they belong to objects which were not created in the pre-state, are excluded. Hence, it reads "subset of the set of those locations" $L^* = \{(o,x) \in \mathcal{L} \mid h_0(o, \texttt{\textbackslash created}) = false\}$ rather than "to equal the empty set". In this limited sense, a weakly pure method does "not assign" any locations.

Another question is, in which state the data groups are determined. This is not made clear in the reference manual. However, since it does clarify this question for method specifications [LPC+08, Sect. 9.6.2], we conform to this point. There, it is said that it is evaluated in the pre-state. Thus, the set L_Λ is defined as $L_\Lambda := loc_{s_0}(\Lambda)$. A visualized example can be seen in Fig. 3.5.

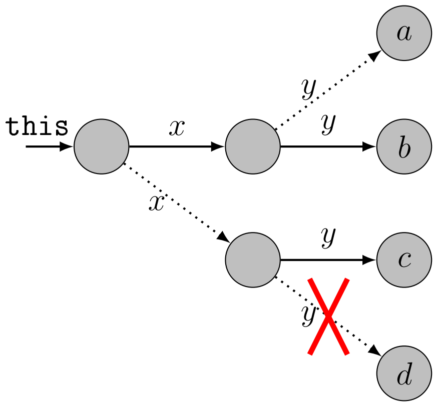

Figure 3.5: Locations for frame conditions are determined in the pre-state. Solid lines represent references in the pre-state, dotted lines in the post-state. The predicate `\only_assigned`$(x, x.y)$ allows $x.y$ to evaluate to a, b or c in the post-state – but not to d.

The `\not_assigned` predicate looks very similar, but it asserts that the locations to which Λ refers were not assigned since the pre-state. In this case, the *intersection* of the set of those locations and the set L_w of assigned locations requires to be a subset of L^*.

$$val_{(s_0,s_1,L_a,L_w,\rho,M)}(\texttt{\textbackslash not\_assigned}(\Lambda)) = true \quad \textbf{iff} \quad L_w \cap \bigcup_{\ell \in L_\Lambda} \mathcal{D}(\ell) \subseteq L^*$$

As already explained above, the `\not_modified` expression is defined differently in that it just compares values of fields. (Where $s_i = (h_i, \sigma_i, \chi_i)$.)

$$val_{(s_0,s_1,L_a,L_w,\rho,M)}(\texttt{\textbackslash not\_modified}(\Lambda)) = true \textbf{ iff}$$
$$h_0(\ell') = h_1(\ell') \qquad \text{for every } \ell' \in \bigcup_{\ell \in L_\Lambda} \mathcal{D}(\ell)$$

For any frame predicate $\mathfrak{F}$ it is $wd_\Sigma(\mathfrak{F}(\Lambda))$ iff $wd_{s_0}(\Lambda)$. And $\omega(\mathfrak{F}(\Lambda)) = \omega(\Lambda)$.

3.3.11 `\fresh`

This predicate is used in postconditions etc. to tell whether a (non-null) object was created since the pre-state. In general, it takes a list of expressions of reference type as arguments. For reasons of simplicity, we consider the case with just one reference expression:

Let $s_0 = (h_0, \dots)$ and $s_1 = (h_1, \dots)$ be system states and x an expression of reference type. $val_{(s_0,s_1)}(\texttt{\textbackslash fresh}(x)) = true$ **iff**

- $val_{s_1}(x) \neq \text{null}$,
- $val_{s_1}(x.\texttt{\textbackslash created}) = true$ and
- $h_0(val_{s_1}(x), \texttt{\textbackslash created}) = false$

Note that in both pre- and post-state, the value of x is determined in the post-state. This ensures that both cases refer to one and the same object.

It is not correct to assert `\fresh(this)` in the post-state of a constructor. As explained in Sect. 2.4.1, a constructor does only *initialize* an already created object. `\fresh` on the other hand yields true if the `new` construct was involved. E.g. the expression `\fresh(new Object())` always evaluates to true.

3.3.12 `\nonnullelements`

This predicate asserts to an array and its members not to refer to null.
$val_s(\texttt{\textbackslash nonnullelements}(a)) = true$ **iff**

- $val_s(a) \neq null$ and
- for every $i \in \{0, \dots, val_s(a.\texttt{length}) - 1\} : val_s(a\texttt{[}i\texttt{]}) \neq null$

Since `Object` is a common super-type for every (Java) array type, it is not statically decidable whether an object reference refers to an array at run-time. Thus, well-definition of this expression depends on whether the referenced element is in fact an array:

$$wd_s(\texttt{\textbackslash nonnullelements}(a)) \textbf{ iff } wd_s(a) \text{ and } val_s(a) \in V_{T\texttt{[]}} \text{ for some type } T$$

3.3.13 Quantifiers

Logical quantifiers

Universal and existential quantification are present in JML. These are always typed and consist of a range expression a, which acts as a guard condition, and a body b:

- $val_{(s_0,s_1)}(\texttt{\textbackslash forall nullable } T\ x; a; b) = true$ **iff** for all $o \in V_T$ it is
$$val_{(s_0^{\{x\mapsto o\}}, s_1^{\{x\mapsto o\}})}(a \texttt{ ==> } b) = true$$
- $val_{(s_0,s_1)}(\texttt{\textbackslash exists nullable } T\ x; a; b) = true$ **iff** for some $o \in V_T$ it is
$$val_{(s_0^{\{x\mapsto o\}}, s_1^{\{x\mapsto o\}})}(a \texttt{ \&\& } b) = true$$

Since both system states s_0 and s_1 are updated, there is no problem with the quantified variable occurring within the scope of `\old`.

Quantification over reference types includes all objects, regardless of being created or not [LPC+08, Sect. 11.4.24.6]. Not created objects have to be excluded explicitly. JML as described in the reference manual yet has got no feature to assert creation. In this thesis, we use the special field `\created`, which is not part of the JML standard.

Per default, the scope of quantification does not include null (as always in JML). For the most general form, we expect them always to be defined `nullable`. If non-nullity is desired, this can be conjoined to the assumptions in the range expression:

$$val(\texttt{\textbackslash forall } T\ x; a; b) = val(\texttt{\textbackslash forall nullable } T\ x; x \texttt{ != } \text{null} \texttt{ \&\& } a; b)$$

Logical quantifiers are well-defined if and only if the expression $(a \texttt{ ==> } b)$ for every valuation of x resp. $(a \texttt{ \&\& } b)$ for one valuation of x is well-defined.[38] In both cases, the range expression a guards the body b from undefinedness.

[38]Following our definition of the conjunction operator, the sub-expressions need to be well-defined in order for the compound expression to valuate to *true*.

One important decision was to disallow quantifiers to have side-effects. Otherwise, we would not only have to consider if they are evaluated in parallel or sequentially (and in what order), but could also create an infinite number of instances. In this case, the successor function ω would not preserve the finite state invariant. Consider for instance the following expression:

```
(\exists Object x; true; x == new Object())
```

In our formalization, this expression would create a new object for every valuation of x. This is done in parallel with every case valuating in the same state. But any information about newly created objects will be dropped after the evaluation. By the way, it will yield true since there exists an object which is the next one to be created.

Generalized quantifiers

JML features five "generalized" quantifiers which respectively yield the sum, product, maximum or minimum of a finite set of numbers as well as the size of a finite set. Although there do exist values (not only limits) for certain infinite sets (e.g. $\prod_{x\in\mathbb{Z}} x = 0$), we restrict every expression to finite sets for reasons of uniformity:

$wd(\mathcal{Q}\ \texttt{nullable}\ T\ x; b; e)$ **iff** $wd(b)$ for every value of x, $wd(e)$ and $|Z| < \infty$ for every generalized quantifier $\mathcal{Q}$ where $Z := \{z \in V_T \mid val_{s\{x\mapsto z\}}(b) = true\}$.

Now for the values. As the expression possibly does not have a value in the mathematical for an infinite set Z, these values remain underspecified. Let us now assume Z to be finite. There is another problem with sum and product since they are determined using arithmetic of the body type. For `\bigint` and `\real` this is just mathematical arithmetic. For integral types the respective modulo arithmetic is in force. Since addition and multiplication are commutative and associative for integral types (i.e. they form commutative rings, e.g. $V_{\text{int}} \cong \mathbb{Z}/2^{32}\mathbb{Z}$), this is well-defined:

$$val_s(\texttt{\textbackslash sum nullable}\ T\ x; b; e) = \bigoplus_{z\in V_T} val_{s\{x\mapsto z\}}(b\ ?\ e :\ 0)$$

Where $\bigoplus$ denotes addition modulo 2^n to distinguish it from the unbounded sum. Likewise $\bigotimes$ denotes multiplication modulo 2^n.

For floating point types, such a definition would not be well-defined as they are not associative due to the lack of precision. E.g.

$$val(\texttt{(pow(2.0,24.0) +1) -1}) = 2^{24} - 1$$

while

$$val(\texttt{pow(2.0,24.0) + (1 - 1)}) = 2^{24}$$

in type float. One approach would be to consider the range of sum and product to be ordered. This approach is very fragile since semantics would heavily depend on that order. We decided to calculate sum or product according to mathematical rules first, and then to cast the result back to the desired floating point type. This means the exact mathematical result to be rounded or to be set to $\pm\infty$:

$$val_s(\texttt{\textbackslash sum nullable } T\ x; b; e) = cast(\texttt{\textbackslash real}, T', \sum_{z \in V_T} val_{s\{x \mapsto z\}}(b\ ?\ e\ :\ 0))$$

The `\max` and `\min` quantifiers return the maximum resp. minimum of the (finite) set $\{val_{\omega(b)(s^{\{x \mapsto z\}})}(e) \mid z \in Z\}$ if it is not empty. For an empty set, JML defines `\max` to be the minimum of values of the body type, `\min` resp. the maximum.[39] E.g.

$$val(\texttt{\textbackslash forall int x; false; x}) = -2^{31}$$

This is however only well-defined for finite numerical types, not for `\bigint` and `\real`. In those cases, we have to extend the well-definition function:

$wd(\mathcal{Q}\ \texttt{nullable}\ T\ x; b; e)$ **iff** $wd(b)$ for every value of x, $wd(e)$, $|Z| < \infty$ and if $T' \in \{\texttt{\textbackslash bigint}, \texttt{\textbackslash real}\}$ then $Z \neq \emptyset$ for $\mathcal{Q} \in \{\texttt{\textbackslash max}, \texttt{\textbackslash min}\}$

To ω the same considerations as for logical quantifiers apply. Thus $\omega(\mathcal{Q}\ \texttt{nullable}\ T\ x; b; e) = id$ for any generalized quantifier $\mathcal{Q}$.

[39]It does not seem very reasonable to assign definite values to undefined cases. In particular, this does not conform overall JML policy. However, this interpretation is disputed and might soon be changed.

Chapter 4

The Meaning of JML Specifications

This chapter covers the different elements of specification in JML – (class) invariants and history constraints, method contracts and general (in-code) assertions. The structure of this chapter is mostly based on the syntactical context in which specifications are given. Invariants and history constraints are defined for classes and interfaces and thus for all methods of this class or interface. Method contracts specify the behavior of one certain method. And assertions are given somewhere within the runnable code.

It is not trivial to discuss them separately, since they heavily depend on each other. As it will be discussed in each section, the meaning of all specification elements is given in the context of methods. This means the definition of all specification elements depends on method specifications (particularly preconditions) while they in turn depend on class invariants.

In the context of verification, for a method to be totally correct with respect to its specification, it has not only to fulfill its method specification, but also to respect the invariants and history constraints of the program and let every assertion hold in its respective states. *Total correctness* of the program Π is given if every method of Π is totally correct. The definitions in this chapter however take no assumptions on the correctness of other methods.

Sect. 4.3.1 describes an approach to handle the values of ghost fields, i.e. fields which are added to the purpose of specification. Although it is very fundamental to this work as a whole, we decided to include it in the section on annotations.

4.1 Class and interface specifications

This section focusses on type, i.e. class or interface, specifications. This includes both invariants and history constraints, which describe a relation between two states. A third type specification element, the `initially` clause will be described in the follow-up section on method specifications, since it only specifies postconditions for constructors. In JML, invariants and history constraints are meant to hold in certain states, called *visible states*. The first subsection covers their definition and a treatise on how to deduce visibility from the run of an execution.

4.1.1 Visible states

Although invariants are always specified within a class or interface, their effective scope is global. E.g. a method m in a class C is obliged to respect invariants of class D.

An exception are *helper methods*. They are thought to do very little computation and can therefore be freed from respecting invariants.[40] Any method or constructor can be declared helper in JML without any further requirement. Since helper methods are only important for the determination of visible states, anywhere else throughout this paper no distinction will be made.

While in some contexts the semantics of invariants are based on *observable states* [BHS07, Sect. 8.2], JML uses the notation of *visible states* [Poe97], which are certain states reached throughout the execution of a code fragment. These two approaches are based on different paradigms. A principle difference is that visible states are not necessarily meant to be visible to an observer, but rather to semantical objects of the program. The targets of visibility, i.e. the objects for which a state is visible, are determined from the running execution and its receivers. The rationale behind that is, that it is primarily intended to impose strong invariants,[41] i.e. which are obliged to hold *every* intermediate state, but secondarily to allow temporary violations of invariants if the “violated object” is a current method receiver. Following the observable state approach, invariants which hold at the beginning of a method invocation also hold at the end. This means that the exact pre- and post-states are the only states *observable*. Visible states however are intermediate states in this sense. According to [LPC$^+$08, Sect. 8.2] they are

[40]However, they are still obliged to fulfill their method contract.

[41]This thought applies to history constraints likewise.

defined as follows.[42]

Definition 4.1 (Visible states (informal))**.** A state is visible for an object o if it is reached at one of the following moments in a program's execution:

1. at the end of a constructor invocation which is initializing o,
2. at the beginning and the end of a non-static (non-helper) method invocation with o as the receiver, i.e. a method like $o.m$ is called,
3. at the beginning and the end of a static (non-helper) method invocation of a class C with $o \in V_C$,
4. when none of the aforementioned invocations is in progress.

The crucial one seems to be the last item. That could be seen overly strict as it seems to require us to check the invariants of nearly every object in every state reached throughout execution. But if we consider a situation in which a class declaration contains public fields, it is desirable to secure they are not arbitrarily changed.

```
1      public class Invariants {
2          private int z = 1;
3          //@ private invariant z > 0;

5          public void a() {
6              z++= 0;
7          }

9          public void b() {
10             z = 0;
11             a();
12         }
13     }
```

Figure 4.1: Example illustrating different invariant semantics

Example 4.2. Fig. 4.1 illustrates the scope of different invariant semantics. The invariant in line 3 requires the value of z to be positive. Obviously, after every execution of a or b the value of z is equal to 1. They both preserve the invariant. Line 11 however corresponds to a visible state since there is a

[42]The case of finalizers however is omitted. See Appendix B.

method invocation. At that point of execution the value of z equals zero, so the invariant is violated after all.

Line 6 also corresponds to a state in which z equals zero. This state is however not visible since it does not involve a method call but the incrementation primitive.

Intermediate states of a constructor are visible

It may appear at the first look as if the first three cases of Definition 4.1 are reducible to the last case. This is not correct: Any method may invoke another one with identical receiver. In this example, the second case applies, while the forth does not. Even the first case (constructor) has to be treated separately because a constructor might invoke another constructor. The post-state of this second invocation is visible to the object which is being initialized. An example is shown in Fig. 4.2. Since `A`'s constructor's post-state is visible, `B`'s invariant is missed.

```
public class A {
    public A ();
}

public class B extends A {
    private /*@ spec_public @*/ int z;
    //@ public invariant z > 0;

    public B () {
        super();
        // visible state for this
        z = 42;
    }
}
```

Figure 4.2: Example with invariant not established by the constructor of the super-class.

Although there is no problem in defining formal semantics, this is a serious problem in praxis. It would imply for virtually every constructor to break its invariant. If not declared explicitly, like in our example, Java enforces calls to `super()` to happen first on every constructor invocation. Even if the super-class constructor establishes *its* invariant, it has no knowledge of fields in sub-classes which need to be assigned to establish the invariant of the *sub-class*. But according to Definition 4.1, its post-state (which is an interior state of the sub-class constructor) is visible for the to-initialize object – even though it is yet not fully initialized.

There are several approaches to deal with this design flaw:

1. Declare every (super-class) constructor which may be called by another one `helper`. This is much like breaking a fly on the wheel as it would free virtually any constructor from establishing its own invariant. In addition, this would require much specification work, so this is not feasible at all.

2. Leavens [Lea06, Lea09] proposed to implement a *guard condition* similar to the one according to the *Boogie* methodology [BDF+04]: Every class definition contains a ghost field `valid` of dynamic type `Object`, which is set by every constructor to some *direct* instance. (See example in Fig. 4.3.) The invariant is then being guarded by an expression which assumes that this field has static type of the sub-class:

   $$Inv \quad \rightsquigarrow \quad \texttt{this.valid instanceof \textbackslash typeof(this) ==>}\ Inv$$

   As in our example, in the post-state of `A`'s constructor, `valid` has dynamic type `A`, which is not a subtype of `B`, so the invariant is trivially true. Although this approach looks quite elegant, this is still a workaround on the specifier level. I.e. every constructor and invariant has to be modified. (It could be done automatically by some tool, however).

3. As a variant of the first approach, one could think of `super` invocations being implicitly declared helper. As it will be described below, our formalization using call stacks of a certain run can be slightly modified to exclude post-states of “inner” constructors. This would mean to modify the official visible state semantics.

4. The most simple approach would be to exclude post-states of constructors at all from the definition of visible states. Since only the “out-most” constructor has to establish the invariant of its class, it is sufficient to assert it in the post-state. All we have to do after all, is to conjoin the invariant to the constructors postcondition as we will do anyway in Sect. 4.2. A state, in which a constructor invokes a (non-constructor) method will be still visible, however. The disadvantage of this approach is that this interpretation does not conform official JML specifications. Since those are in our opinion inappropriate to use in any specification, we regard our deviation justified.

```
public class A {
    //@ public ghost Object valid;
    //@ private final static ghost A validForA;
    public A () {
        //@ set valid = validForA;
    }
}

public class B extends A {
    //@ private final static ghost B validForB;

    private /*@ spec_public @*/ int z;
    /*@ public invariant valid instanceof \typeof(this)
      @      ==> z > 0;
      @*/

    public B () {
        super();
        //@ assert \typeof(valid) == \type(A);
        z = 42;
        //@ set valid = validForB;
    }
}
```

Figure 4.3: Guarded invariants as proposed with approach 2.

The static case

According to [LPC$^+$08], a state is visible for a type T (i.e. class or interface) if it occurs after static initialization of T and it is a visible state for some object of static type T. Leaving out static initialization – which we do in this paper – this would lead to the fact that *every* state reached by the virtual machine is visible to every class and interface. This is because there is always an infinite number of instances for which the fourth case of Def. 4.1 applies. So, this definition would be too strong as intermediate states of a static method invocation would be visible.

On the other hand, this definition would be too weak in some cases. Consider an abstract class with static fields but no concrete sub-classes and thus no instances. In this case, no state would be visible since there are no instances for which it would be visible.

To overcome this, we will use another definition for visibility which is not based on visibility for instances, but rather constructed dually:[43]

Definition 4.3 (Visible states (informal, static case))**.** A state is a visible state for a type T if it is reached at one of these moments in a program's execution:

- at the beginning or end of a non-helper method invocation of a static method of T or a sub-type of T or
- when no static method of T or a subtype of T is in progress, and when its visible for all instances of T.

A formalization of visible states

The above definition of visible states can be further refined using the definition of system states. We do not only inspect the state in question but a whole run in which it occurs. An example can be seen in Fig. 4.4. The first three items in following enumeration are meant to mirror the cases of Definition 4.1 (excluding constructors as discussed above), while 4 and 5 cover the static case (Def. 4.3). Let M_C be the set of all non-helper methods of class C or its super-classes.

Definition 4.4 (Visibility relation)**.** Let $R = \langle \ldots, s_i, s_{i+1}, \ldots \rangle$ be a run containing at least two states (with $s_j = (h_j, \sigma_j, \chi_j)$). The *visibility relation* $\mathcal{V}_R \subseteq R \times (\mathcal{U} \cup \mathcal{C})$ is given as the least set satisfying:

[43]This definition has yet not been accepted to JML, but has been discussed in [Lea09].

1. $(s_i, o) \in \mathcal{V}_R$ if $\mathrm{top}(\chi_1) = (\mu, o)$ for a (non-helper, non-constructor, non-static) method μ and $\mathrm{pop}(\chi_1) = \chi_i$ (beginning of invocation on o). $(s_{i+1}, o) \in \mathcal{V}_R$ in the dual case (end of invocation on o).

2. $(s_i, o) \in \mathcal{V}_R$ if $\mathrm{top}(\chi_{i+1}) = \mu$ for a (non-helper) static method μ in C, $\mathrm{pop}(\chi_{i+1}) = \chi_0$ and $o \in V_C^-$ (beginning of static invocation). $(s_{i+1}, o) \in \mathcal{V}_R$ in the dual case (end of static invocation).

3. $(s_j, o) \in \mathcal{V}_R$ if $o \in \bigcup_{C \in \mathcal{C}} \{u \in V_C^- \mid \forall \mu \in M_C.((\mu, u) \notin \chi_j \wedge \mu \notin \chi_j)\}$ (no invocation in progress)

4. $(s_i, C) \in \mathcal{V}_R$ if $\mathrm{top}(\chi_{i+1}) = \mu$ for a (non-helper) static method μ in $D \sqsubseteq C$, $\mathrm{pop}(\chi_{i+1}) = \chi_0$ (beginning of static invocation). $(s_{i+1}, C) \in \mathcal{V}_R$ in the dual case (end of static invocation).

5. $(s_j, C) \in \mathcal{V}_R$ if $(s_j, o) \in \mathcal{V}_R$ for every $o \in V_C^-$ and for every static method μ of a class $D \sqsubseteq C$ it is $\mu \notin \chi_j$ (no static invocation in progress).

The JML reference manual does not clarify whether states can be visible to objects which are not created. We decided to have all objects (except null) included. Of course, an instance invariant only makes sense when applied to created objects. As a “corrective”, we will later restrict the definition of an instance invariant to created ones. In the aforementioned case in which a type T has absolutely no instances, states are visible for T if and only if no static method of T is in progress.

Theorem 4.5 (Visible state theorem)**.** *A state s is visible to an element $x \in \mathcal{U} \cup \mathcal{C}$ if and only if $(s, x) \in \mathcal{V}_R$ for some run R with $s \in R$.*

4.1.2 Invariants

One of the most important and widely-used specification elements in object-orientation are type invariants. These can be seen as conditions to constrain the state an instance can be in. But as we have discussed above, there is in fact no “the” invariant to be preserved by methods, but rather *all* invariants of the program. At first, we need to capture all invariants which are applicable to the class or interface of concern. There may be several invariant definitions which must all be respected, so they are equal to a single invariant consisting of a conjunction of the former. And of course, invariants may be inherited, too.

```
public class CoffeeMaker {
    private Water w;
    private Beans b;

    public Coffee make() {
        if (!empty()) {
            w.boil();
            return new Coffee (w, b);
        }
    }

    protected boolean empty () {
        return w == null || b.grind() == null;
    }
}
```

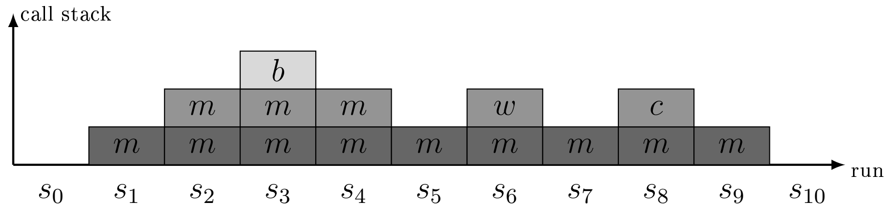

Figure 4.4: An example for visible states deduced from the call stack. Consider the program code above and objects m, w, b, c of types `CoffeeMaker`, `Water`, `Beans` and `Coffee` respectably. The changing call stack of the run of `make` is shown below.
For w, b and c all states are visible, except those in which they act as receiver (s_3, s_6, s_8 respectively). s_0 – s_7 are visible for c, too, though it is not yet created. For m only the pre- and post-states of method invocations are visible: s_0/s_{10} and s_1/s_5.

Augmented invariant

In the following, we collect all of these to a single formula which will be called the *augmented invariant* of class C. Consider a class (or interface) $C = C_0$ with the following specifications where M_i are privacy and static modifiers:

```
/*@ M_01  invariant  expr_01;
  @ ...   ...
  @ M_0n  invariant  expr_0n;
  @*/
```

Then, the short-circuit conjunction over all $\texttt{expr}_{0i}$ is part of the augmented invariant of C. C also inherits all non-static[44] invariants from its super-classes and implemented interfaces if they possess at most protected privacy or they possess default privacy and are declared in a class/interface which is contained in the same package as C. Private and static invariants are never inherited. Since invariants protect each other against undefinedness (see [LPC+08, Sect. 2.7]), their exact order does matter.[45]

Let $(\mathcal{T}, \preceq)$ be a linear type hierarchy (see Def. 2.13) and $C_0 \prec C_1 \prec \ldots \prec C_k$ the complete ascending chain from $C = C_0$ on. Let the inherited expressions of C_j be $\texttt{expr}_{j1}, \ldots, \texttt{expr}_{jn_j}$. The augmented invariant of C_0 finally is the (short-circuit) conjunction of all those properties in the following order:

$$I_C^{\text{aug}} := (\texttt{expr}_{k1}\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \texttt{expr}_{kn_k}\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \texttt{expr}_{01}\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \texttt{expr}_{0n})^{\mathcal{N}}$$

Note that the augmented invariant also contains static invariants of class C. This is not problematic since we defined static fields to be contained in every instance of the class (see Sect. 2.3). The normalized (short-circuit) conjunction of all static invariants of C is denoted by I_C^{static}. Note that even though there are no applicable direct instances, interfaces and abstract classes may be annotated with instance invariants.

Meaning of invariants

The meaning of invariants is that they hold at every state which is visible to the object or type of concern. This can be broken down to the requirement that invariants are respected by all available methods, i.e. if the invariant and at least one of the method's augmented preconditions (see Sect. 4.2.2) hold

[44]At the time of writing it was still an open discussion whether static invariants and (history constraints) are inherited. Effectively, this does not make a great difference.

[45]Although it is not mentioned in the reference manual, we expect that invariants specified in super-classes protect those of sub-classes.

in the pre-state, then the invariant must hold in every visible state reached by the method.

An interesting consequence of visible state semantics is that it is determinable from the given system state which invariants hold. Since we require the preservation of invariants in every possible pre-state, we essentially require that *every combination* of invariants is preserved by the method. E.g. if no other method is currently running (i.e. on the call stack), all invariants of every object or type are applicable. Every method, which resides additionally on the call stack, again reduces the number of applicable invariants.

Definition 4.6 (Respect)**.** Let μ be a [non-]static method with body π, [receiver object r] and an augmented precondition $\overline{pre}$. Let $M \in \mathcal{M}$ be a model field valuator and $s \in \mathcal{S}$ a system state with [$val_s(\texttt{this}) = r$ and]

$$(s, s, L_a, L_w, M, \rho)) \models \overline{pre}$$

Let the black box execution $\mathcal{J}(s, \pi)$ yield a run R.

- Let for every class or interface $C \in \mathcal{C}$ with $(s, C) \in \mathcal{V}_R$ the augmented static invariant hold:

$$(s, s, L'_a, L'_w, M, \rho')) \models I_C^{\text{static}}$$

- Let for every (non-abstract) class $C \in \mathcal{C}$ and every $o \in V_C^0$ with $(s, o) \in \mathcal{V}_R$ the augmented instance invariant hold:

$$(s', s', L'_a, L'_w, M, \rho')) \models I_C^{\text{aug}}$$

  with s' coinciding with s except for $val_{s'}(\texttt{this}) = o$ and $val_{s'}(\texttt{this.\textbackslash created}) = true$ and $L_a, L'_a, L_w, L'_w, \rho, \rho'$ are arbitrary.

μ *weakly respects* the invariants of Π if, in every state $s^* \in R^*$

- with $(s^*, C) \in \mathcal{V}_{R^*}$ the augmented static invariant holds for every $C \in \mathcal{C}$:

$$(s^*, s^*, L''_a, L''_w, M, \rho'') \models I_C^{\text{static}}$$

- and for every $o \in V_C^0$ with $(s^*, o) \in \mathcal{V}_{R^*}$ the augmented instance invariant holds:

$$(s'', s'', L''_a, L''_w, M, \rho'') \models I_C^{\text{aug}}$$

  where s'' coincides with s^* except for $val_{s''}(\texttt{this}) = o$ and $val_{s''}(\texttt{this.\textbackslash created}) = true$ and L''_a, L''_w, ρ'' are again arbitrary.

μ *respects* the invariants of Π, if it weakly respects them for every reachable pre-state $s \in \mathcal{S}$ and model field valuator $M \in \mathcal{M}$.

R^* denotes the updated run of Sect. 4.3.1. We are neither interested in accessed/assigned locations nor the return value of any logical state since neither storage expressions nor the result expression are allowed in preconditions and invariants. In addition, we evaluate only with one state since the `\old` clause would not make sense and thus is ignored. We made no mention of parameters to the method μ since the set of all possible states already contains the values for all possible parameters.

Note that the kind of termination of a method does not matter. Regardless of terminating normally, exceptionally or erroneously, a method has to meet the invariant in every visible state. The Reference Manual does not say anything regarding errors. One could postulate the specification to be met trivially in this case, as this would be an analogy to method specifications (see Sect. 4.2). We however assume stronger invariant semantics. (After all, an error is an error and, it may be caused by bad programming or specifications which could be uncovered.) The same consideration applies to non-termination, but in this case there may be infinitely many visible states.

4.1.3 History constraints

History constraints [LW93] are in a way similar to invariants as they constrain the state which an object may be in. But while invariants must hold for every visible state, history constraints describe the relation of two visible states following each other throughout a possible program execution. History constraints may rely on syntactical features which are used to measure changes between states such as the `\old` clause as well as frame expressions.

Similar to invariants, there may be several constraint definitions and non-private constraints are be inherited. Therefore we introduce an *augmented history constraint* H_C^{aug} and a static variant H_C^{static} which are constructed analogously to above.

Definition 4.7 (Respect)**.** Let μ be a method with body π, receiver object r if it is non-static, and an augmented precondition $\overline{pre}$. Let $M \in \mathcal{M}$ be a model field valuator and $s = (h, \sigma, \chi) \in \mathcal{S}$ a system state with $val_s(\texttt{this}) = r$ (if μ is non-static)

$$(s, s, L_a, L_w, M, \rho)) \vDash \overline{pre}$$

where L_a, L_w, ρ are unknown but fixed. Let the black box execution yield $\mathcal{J}(s, \pi) = (R, \lambda_a, \lambda_w, \alpha, \Omega, \nu)$.

μ *weakly respects* the history constraints of Π if,

- for every $C \in \mathcal{C}$ and every ordered pair of states $s_i, s_j \in R^*$ with $i \leq j$ and with $(s_i, C), (s_j, C) \in \mathcal{V}_{R^*}$, it holds that

$$(s_i, s_j, \lambda_a(i,j), \lambda_w(i,j), M, \rho') \models H_C^{\text{static}}$$

- and for every $o \in V_C^0$ and every ordered pair of states $s_{i'}, s_{j'} \in R^*$ with $i' \leq j'$ and with $(s_{i'}, o), (s_{j'}, o) \in \mathcal{V}_{R^*}$, it holds that

$$(\tilde{s}_{i'}, \tilde{s}_{j'}, \lambda_a(i',j'), \lambda_w(i',j'), M, \rho') \models H_C^{\text{aug}}$$

  where $\tilde{s}_k$ coincide with s_k except for $val_{\tilde{s}_k}(\texttt{this}) = o$ and $val_{\tilde{s}_{i'}}(\texttt{this.\textbackslash created}) = \mathit{true}$ and ρ' is arbitrary.

μ *respects* the history constraints of Π if it weakly respects them for every system state $s \in \mathcal{S}$ and model field valuator $M \in \mathcal{M}$.

In particular, the pre-state s may occur as first argument of the considered pairs of states and, if the method terminates, the post-state as second argument. Thus it is a special case of the meaning of the constraint that it relates pre- and post-condition.[46]

The JML reference manual designates history constraints to denote reflexive and transitive relations. Both characteristics are sensible since it is non-trivial for an observer to deduce which states are visible. Transitivity immediately follows from the requirement that history constraints are defined for every pair of states, rather than just two succeeding states. Reflexivity is enforced by the reflexive relation $\leq$ on R^*.

4.2 Method specifications

Method specifications, or *method contracts*[47], are primary means for the specification of the behavior of a specific method. In JML, there is large variety of method specification clauses. It does not conform a classical “precondition/-postcondition schema” of Hoare logic [Hoa69]. Instead, it can be specified both normal termination, abrupt termination by the throw of an exception and non-termination. To any of these cases there exists a corresponding condition. In this way, JML specifications may be more specific than formulae in dynamic logic or constraints in OCL. In addition, several frame properties can be specified which have to hold in the case of termination.

[46]The JML reference manual [LPC$^+$08, Sect. 8.3] does only call this relation “respect”, but we extended this notation to every visible state.

[47]We use these terms synonymously.

In this section, we develop a formalization from the informal description of method specifications given in the JML reference manual [LPC+08, Sect. 9.6.3]:

> "Consider a particular call of the method m. The state of the program after passing parameters to m, but before running any of the code of m is called the *pre-state* of the method call.
>
> Suppose all applicable invariants hold in the pre-state of this call. [...] Suppose also that [...] the precondition, P, from the `requires` clause, holds. [...] Then one of the following must also hold:
>
> - the `diverges` predicate, D, holds in the augmented pre-state and the execution of the method does not terminate [...] or
> - the Java virtual machine throws an error [...], or
> - the method terminates by returning or throwing an exception, reaching a state called its post-state, in which all of the following hold. [...]
>   - During execution of the method (which includes all directly and indirectly called methods and constructors), only locations that either did not exist in the pre-state, that are local to the method (including the method's formal parameters), or that are either named in the lists R and A found in the `accessible` and `assignable` clauses or that are dependees [...] of such locations, are read from. [...]
>   - During execution of the method, only locations that either did not exist in the pre-state, that are local to the method, or that are either named by the `assignable` clause's list, A, or are dependees [...] of such locations, are assigned to. [...]
>   - If the execution of the method terminates by returning normally, then the normal postcondition, Q, given in the `ensures` clause, holds in the post-state.
>   - If the execution of the method terminates by throwing an exception of some type E_a that is a subtype of `java.lang.Exception`, then: [...] if E_a is a subtype of the type E given in the `signals` clause, then the exceptional postcondition R must hold in the post-state,

augmented by a binding from the variable e to the exception object thrown.

– All applicable invariants and history constraints hold in the post-state."

Exceptions and errors

Programs may throw exceptions and errors in some kind of abnormal execution. Java traditionally discriminates between *exceptions* and *errors*. While the former is meant to be caught by some handler within the program, the latter is reserved for most severe problems. Errors are not meant to be caught but to terminate the whole program abruptly. A typical example for an error is that the (physical) machine runs out of memory.

In the JML view of things, an execution which terminates by a thrown exception is still within the scope of specification considerations. It is distinguished between normal and exceptional post-conditions (see Sect. 4.2). This is perfectly reasonable since it is deterministic whether an exception is thrown or not.

This is not true for errors. The JML reference manual defines any method contract to be met if the method terminates with an error [LPC+08, Sect. 9.6.2]. On the one hand, errors may appear in an unpredictable manner (and thus in some sense nondeterministically), so it may be justified to ignore them. On the other hand, an error represents a severe failure of the software system and must not be overlooked [Blo01].

One pathological example would be the difference between assertions in JML (within Java comments) and the assert statement in Java. If the property is not met at the desired program point, the JML assertion is not valid. Whereas an assertion which is written directly within Java code would throw an `AssertionError`. As an alternative, it has been proposed that JML should treat the throw of an error as a kind of non-termination [Eng05]. For this work however, we try to stick to the reference manual as close as possible.[48]

Non-null by default

By default, JML excludes null references from the scope of reference type fields, variables, parameters and return values. While this definition differs from Java, it has been argued [CR06] that it is a most common programming (and specification) error to forget that null is included with every static type.

[48]Nevertheless, we restrict our machine model to not throwing any unpredicted errors. (See Sect. 2.4)

In this work we will almost always consider the more general form, including null. This means every reference (of parameters and return values) has to be declared with the keyword `nullable`. The non-null default can be seen as syntactical sugar for a constraining expression in class invariants or method contracts.

4.2.1 Desugaring method specifications

To begin with, we need to find a suitable syntactical form to define semantics on. There may be several pre- and post-conditions as well as exception specifications. To make things more complicated, method specifications may be nested, too. This emerges the need for a "canonical", de-sugared form of method specifications. [RL00] presents an extensive 11-item list of substitutions. Following this approach thoroughly might render the specification not very readable, however. We assume the following steps:

1. Modify every reference type occurring as (explicit) parameter or return type with `nullable` and desugar `non_null` (i.e. default) modifiers.
2. Desugar `pure` modifiers to `requires true`, `diverges false` and `assignable \nothing`.
3. Eliminate nested specifications from the inside out.
4. Desugar different behavioral cases (e.g. `normal_behavior` to `behavior` with adding `signals false` and `diverges false`).
5. Standardize `signals` and `signals_only` clauses to just one `signals` clause with `Exception` as caught exception type.
6. Conjoin (short-circuit) multiple clauses of the same kind (e.g. `requires`).

This roughly corresponds to the first nine steps of [RL00], but does not conjoin specification cases (i.e. blocks beginning with `behavior`). An example can be seen in Fig. 4.5.

This leads to a number of canonized *specification cases* as displayed in Fig. 4.6. Where M_i is a privacy modifier, and the `requires`, `ensures`, `signals` and `diverges` clauses accommodate boolean expressions. A_r and A_w are lists of reference expressions. We further require all clauses to be fully specified[49] and normalized. We will denote the specification case by the

[49]Although JML allows the keyword `not_specified`, its interpretation is meant to be determined by an implementing tool, not by the language specification.

```
/*@ public normal_behavior
  @     requires o instanceof Float;
  @     {|
  @          requires   o.value() != Float.NaN;
  @          ensures    \result.length() > 3;
  @       |}
  @
  @ also
  @
  @ public exceptional_behavior
  @     requires          !(o instanceof Float);
  @     signals (UnsupportedOperationException e) true;
  @     signals_only   RuntimeException;
  @*/
public static String toString (Object o);


/*@ public behavior
  @     requires   o != null && o instanceof Float
  @                 && o.value() != Float.NaN;
  @     ensures    \result != null
  @                 && \result.length() > 3;
  @     signals (Exception e) false;
  @     diverges    false;
  @
  @ also
  @
  @ public behavior
  @     requires   o != null && !(o instanceof Float);
  @     ensures    false;
  @     signals (Exception e)
  @          (e instanceof UnsupportedOperationException)
  @              ==> true
  @          && !(e instanceof RuntimeException)
  @              ==> false;
  @     diverges    false;
  @*/
public static /*@ nullable @*/ String
        toString (/*@ nullable @*/ Object o);
```

Figure 4.5: Method specification with syntactical sugar (above) and a semantical equivalent "diet" version (below).

```
/*@ $M_i$ behavior
  @      requires   $pre_i$;
  @      ensures    $post_i$;
  @      signals (Exception e) $xpost_i$;
  @      diverges   $dpre_i$;
  @      accessible $A_r$;
  @      assignable $A_w$;
  @*/
```

Figure 4.6: The canonized form of a method specification case.

tuple $B_i := (pre_i, post_i, xpost_i, dpre_i, A_{ri}, A_{wi})$. The set of all specification cases for a given method is called the *method contract*.

As with all other specification elements, method specification cases are inherited according to Java rules (see also Sect. 2.5). Since these specifications are attached to certain methods in super-classes, it is not completely trivial to decide which specifications a method inherits. It does inherit exactly those inheritable method specification cases which are annotated to a method which is overridden, i.e. a method which possesses an identical signature.

4.2.2 Method specifications formally

Augmented precondition

Coming up next, it is to determine what is meant by "all applicable invariants". As it was described in Sect. 4.1.1, invariants are applicable if and only if the state is visible for an instance or type, which is specified with that invariant. But for which of them is the pre-state visible? As a complication, we defined visible states only within certain runs. On a closer look however, there is only one (augmented) invariant which can be assumed applicable. The method specification has to hold for every system state which qualifies as pre-state (i.e. all parameters (including `this`) valuate to the values passed to the method). Thus, in the worst case, all created objects are receivers of methods on the call stack, and there is a static method for every class on the call stack. This means that the pre-state is only visible for the receiver object of the method of concern itself (resp. for the class if the method is static).

Although the reference manual is not clear on that point, we assume that invariants protect the precondition from undefinedness. (Quite typical an invariant specifies a field to be non-null and all method specifications rely on this, for instance.) Of course, a constructor cannot assume any invariant.

To capture applicable invariants, we define the *augmented precondition* $\overline{pre}$, given a method μ with precondition pre, as

$$\overline{pre} := \begin{cases} pre & \mu \text{ is a constructor} \\ I_C^{\text{aug}} \text{ \texttt{\&\&} } pre & \mu \text{ is a non-static method with receiver } o \in V_C^0 \\ I_C^{\text{static}} \text{ \texttt{\&\&} } pre & \mu \text{ is a static method in class } C \end{cases}$$

Augmented postcondition

Parameters are allowed to appear in both normal and exceptional postconditions. In specification, they are always meant to refer to the values which are passed to the method. (Those are not mentioned here since they are implicitly included in the pre-state's Stack.) In execution however, they are local to the method and their values may be changed. Therefore, parameters are always evaluated in the pre-state [LPC$^+$08, Sect. 9.9.6]. We will syntactically replace all parameters p with `\old`(p).

We also have to ensure that if the method terminates (normally or exceptionally), then "all applicable" invariants[50] hold. We could in fact try a more modular approach and not require invariants to hold in the post-state as it is a visible state (see Sect. 4.1.1). However since we left out constructors from the definition of visible states (Def. 4.4), this has already become necessary (at least for constructors).

With constructors always comes another JML-speciality: `initially` clauses. Syntactically these are type specifications, but in fact denote additional post-conditions for all constructors of the specified type. `initially` clauses are inherited just like invariants. Let $init_C$ be the accordingly constructed (with respect to some linear type hierarchy (see Definition 2.13)) short-circuit conjunction of all `initially` clauses defined in super-types of class C and inherited according to Java rules. Note that even interfaces can be specified with `initially` clauses even though they do not possess constructors.

To include invariants and valuate parameters in the right state, we define the *augmented postcondition* $\overline{post}$, given a method μ with parameter identifiers $p_1, \ldots, p_n$ and a (normal or exceptional) postcondition $post$, as

$$\overline{post} := \begin{cases} post' \text{ \texttt{\&\&} } I_C^{\text{aug}} \text{ \texttt{\&\&} } init_C & \mu \text{ is a (non-helper) constructor in class } C \\ post' \text{ \texttt{\&\&} } I_C^{\text{aug}} & \mu \text{ is non-static with receiver } o \in V_C^0 \\ post' \text{ \texttt{\&\&} } I_C^{\text{static}} & \mu \text{ is a static method in class } C \end{cases}$$

[50]To be precise, the reference manual speaks of "invariants and history constraints". But since history constraints are no single-state properties, this would make no sense.

where *post'* is constructed from *post* by replacing every occurrence of p_j by `\old`(p_j).[51]

Specification fulfillment

The properties arising from frame conditions share much similarity with their "predicate counterparts" (see Sect. 3.3.10, p. 58), as difference, locations from the `assignable` clause are implicitly included in the list of locations from the `accessible` clause.

We are finally led to the following definition:

Definition 4.8 (Fulfillment)**.** Let μ be a method of class C with code fragment π and $\{B_1, \dots, B_k\}$ its method contract with $B_j = (pre_i, post_i, xpost_i, dpre_i, A_{ri}, A_{wi})$. μ *fulfills* its contract if

in every system state s_0 and for every model field valuator $M \in \mathcal{M}$, and if μ is non-static for every $o \in V_C^0$ with $val_{s_0}(o.\texttt{\textbackslash created}) = true$ and $val_{s_0}(\texttt{this}) = o$, the following holds for all B_i:

If $(s_0, s_0, L_a, L_w, M, \upsilon) \vDash \overline{pre_i}$ holds where L_a, L_w and υ are arbitrary and $\mathcal{J}(s_0, \pi) = (R, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$, then one of the following also holds:

- (Non-termination) $|R| = \infty$ and $(s_0, s_0, L'_a, L'_w, M, \upsilon') \vDash dpre_i$ with L'_a, L'_w and υ' unknown but fixed
- (Erroneous termination) $\Omega \in V_{\texttt{Throwable}} \setminus V_{\texttt{Exception}}$
- (Normal termination)
  - $R^* = \langle s_0, \dots, s_k \rangle$, $\Omega = \text{null}$,
  - $(s_0, s_k, \lambda_a(0,k), \lambda_w(0,k), M, \rho) \vDash \overline{post_i}$
  - $\lambda_a(0,k) \setminus \bigcup_{\ell \in loc_{s_0}(A_{ri} \cup A_{wi})} \mathcal{D}(\ell) \subseteq L^*$ and
  - $\lambda_w(0,k) \setminus \bigcup_{\ell \in loc_{s_0}(A_{wi})} \mathcal{D}(\ell) \subseteq L^*$
- (Exceptional termination)
  - $R^* = \langle s_0, \dots, s_k \rangle$, $\Omega \in V^-_{\texttt{Exception}}$,
  - $(s'_0, s'_k, \lambda_a(0,k), \lambda_w(0,k), M, \rho) \vDash \overline{xpost_i}$ with $s'_j = s_j^{\{e \mapsto \Omega\}}$
  - $\lambda_a(0,k) \setminus \bigcup_{\ell \in loc_{s_0}(A_{ri} \cup A_{wi})} \mathcal{D}(\ell) \subseteq L^*$ and

[51] Since we defined the black box to restore parameters at the end of a method invocation, it is not really necessary to put an explicit `\old`. We however keep it for reasons of clarity.

– $\lambda_w(0,k) \setminus \bigcup_{\ell \in loc_{s_0}(A_{wi})} \mathcal{D}(\ell) \subseteq L^*$

where $L^* := \{(o,x) \in \mathcal{L} \mid h(o, \texttt{\textbackslash created}) = \mathit{false}\}$.

Here, R^* denotes the updated run of Sect. 4.3.1.

In this definition (and the official one in the reference manual), frame properties are only required to hold whenever there is a definite post-state, i.e. upon normal or exceptional termination. For some purposes this condition is too weak. Consider for instance a method which is specified to loop forever – something which is very common in praxis. In this case it would be freed from respecting `assignable` clauses. To some extend, namely if the infinite loop contains visible states, these frame properties can be specified using history constraints. To achieve a more general notion, it has been recently proposed [Leh09] to have those properties included in every termination case. Since the run might be of infinite length, the above definition needs to be generalized in that the condition does hold for every pair of pre-state and any other state within the run, as opposed to just the pair pre-state/post-state.

Example 4.9 (JML and dynamic logic[52]). As it has been explained above, dynamic logic (DL), in contrast to JML method specifications, only discriminates between the case where normal termination is necessary and every other case. The basic DL formula with pre- and postconditions $\mathit{pre} \rightarrow [\pi]\mathit{post}$ can be modeled in JML as:

```
/*@ requires pre;
  @ ensures  post;
  @ diverges true;
  @ signals (Exception e) true;
  @*/
```

If termination is required, both `diverges` and `signals` clauses must carry `false`. Due to some workaround using Java's try/catch blocks, DL is also capable of expressing a method specification where the `signals` clause is not trivially true or false:

$$\mathit{pre} \rightarrow [\texttt{try}\ \{\pi\}\ \texttt{catch(Exception}\ e)]((e = \text{null} \rightarrow \mathit{post}) \land (e \neq \text{null} \rightarrow \mathit{xpost}))$$

For a non-trivial `diverges` clause *div* the DL formula is split into two implications, specifying partial correctness resp. (normal) termination:

$$(\mathit{pre} \rightarrow [\pi]\mathit{post}) \land (\neg \mathit{div} \rightarrow \langle\pi\rangle \mathit{true})$$

[52]See also [Eng05, Sect. 5.1]

4.3 Annotation Statements

Annotation statements are JML statements which may occur anywhere within Java code as a comment. Although the three covered annotation statements serve very different purposes, they are grouped here due to their common appearance. For most generality, an annotation statement is verbatim a keyword (e.g. `assert;`) followed by an appropriate expression. In contrast to all other specification elements, annotations are not attached to some syntactical entity (such as a method declaration), but rather to certain states[53] which occur during the execution of a program. In particular, there is no inheritance of annotations.

Section 2.1 already defined the annotated code fragment π^A with a set A of annotation statements and the definition of the Java black box (2.4) included the relation α. We will use this relation to determine the specified states.

4.3.1 Updating ghost variables and fields

Beneath model fields, JML also features additional specification-only variables and fields, which are used almost like variables and fields declared in the program. In contrast to model fields, these *ghost* variables and fields are explicitly assigned to through a special annotation statement, the `set` statement.

With the exception that ghost fields may appear as members of interfaces, they do not differ syntactically or semantically from their counterparts defined with the program. In particular, they share a common name space. Therefore, we assumed that they are not treated differently in expression evaluation and are likewise addressed via Heap and Stack.

Ghost variables and fields may only be assigned through annotations but not within the program. This means that the black box execution – which is restricted to evaluate Java – is not capable of assigning these values and we need to do this “manually”. On the other hand, execution does not depend on them. So it is possible to assign them a-posteriori. In our approach, we first let the black box yield a complete run and then, we apply the respective assignments on all states of the run. This is done by altering valuation through Heap and Stack.

There may be several annotations to the code fragment such that their position within the code corresponds to the very same state within the run.

[53]Since the JML reference manual does itself not provide a clear definition of a state, it is not clear either which state an annotation according to the official reference denotes.

Since those may be conflicting, i.e. they are not parallelizable, we have to apply them one at a time. A set statement always consists of a Java-like assignment. The right-hand-side must be a pure expression.[54] Other expressions (with impure side-effects) such as x`++` are not allowed.

At first, we need to define how assignments through the `set` statements are applied. Let A, R and α be given as of Sect. 2.4. We define an *application function* $\Theta : A \to (\mathcal{S} \to \mathcal{S})$ which applies a single assignment to a system state.

$$\Theta(\texttt{set x=}v\texttt{;})(s) := \begin{cases} \omega(v)(s)^{\{x \mapsto val_s(v)\}} & x \text{ variable} \\ \omega(v)(s)^{\{loc_s(x^{\mathcal{N}}) \mapsto val_s(v)\}} & x \text{ non-static field} \\ \omega(v)(s)^{\{(o,x) \mapsto val_s(v) | o \in V_C\}} & x \text{ static field of class } C \end{cases}$$

It is $\Theta(\mathfrak{a}) := id$ for any annotation $\mathfrak{a}$ which is not a `set` statement. In the case x denotes a variable, we simply update the Stack through adding the appropriate value. If x denotes a field, which receiver is not necessarily the receiver of the current method, we have to evaluate the location to which x refers to first.

Secondly, we collect all applicable assignments. For every state s_j of the run $R = \langle s_0, s_1, \dots \rangle$ we define the sequence $\mathfrak{A}_s$ of applicable assignments:

$$\mathfrak{A}_{s_j} = \langle \alpha(s_0)(1), \dots, \alpha(s_0)(|\alpha(s_0)|), \dots, \alpha(s_{j-1})(1), \dots, \alpha(s_{j-1})(|\alpha(s_{j-1})|) \rangle$$

This includes all assignments which are associated to states of the run preceding s_j in the exact order in which they appear in the program code. Finally, we retrieve a notion for updating a state, resp. a whole run.

Definition 4.10. Let $s \in R$ be a reachable system state and the elements of $\mathfrak{A}_s$ be given as $\mathfrak{a}_1, \dots, \mathfrak{a}_k$.[55]

1. To s the *updated state* s^* is given by $s^* = (\Theta(\mathfrak{a}_k) \circ \Theta(\mathfrak{a}_{k-1}) \circ \cdots \circ \Theta(\mathfrak{a}_1))(s)$
2. The *updated run* R^* is retrieved from R by replacing every s by its updated counterpart s^*. We also write α^* as short-hand with $\alpha^*(s^*) = \alpha(s)$.

Example 4.11. Consider the code fragment from Example 2.12 annotated with a ghost variable z which counts the number of loop iterations. In order for the loop to be correct, the value of z in the post-state must equal the value of x in the pre-state.

[54] It might however have side-effects.

[55] Since A is always a finite set, every state at a finite position in R, i.e. a reachable state, is updated only a finite number of times. Therefore this definition is well-defined.

```
    public int factorial (int x) {
        int y = 1;
        //@ ghost int z = 0;
        while (x > 0) {
            //@ set z = z + 1;
            y = y * x--;
        }
        //@ assert \old(x) == z || z == 0;
        return y;
    }
```

Every loop iteration corresponds to at least one state s_j in a run $R = \langle s_0, \ldots, s_n \rangle$. WLOG this denotes the state before the control flow enters the loop. Then it is $\alpha(s_0) = \langle(\texttt{set z = 0;})\rangle$ and $\alpha(s_j) = \langle(\texttt{set z = z+1;})\rangle$ for every $j \in [1, n]$. While $val_{s_j}(z)$ is underspecified for every j, i.e. the identifier z is not on the Stack, in the updated run it is $val_{s_j^*}(z) = j$. We can finally prove that $val_{s_0}(x) = val_{s_n^*}(z)$.

Since we have shown that the valuation of every pure expression preserves the reachable state property, every updated state s^* is reachable if and only if the original state s is reachable.

4.3.2 Assertions and assumptions

Assertions [Hoa69] in JML pose yet another, more “low-level”, specification instrument. When the validity of assertions is concerned, there is a question of the context in which they are evaluated. Since they historically are incorporated from run-time checking, one could argue that they should hold in any program run reached from the main method. We instead require a stronger and more modular interpretation: For every legal execution (i.e. a precondition and all applicable invariants hold) of a method, the assertions annotated to the method body must hold.

JML also introduces annotations beginning with the `assume` keyword. These are meant to be dual to assertions, in the sense that assertions have to be verified, whereas *assumptions* are – as the name suggests – assumed to hold. The reference manual does however not give a clear semantics for assumptions [LPC$^+$08, Sect. 12.4.1]. We take the position to see an assumption as a kind of “in-code-precondition”, i.e. if it does not hold, no assertion occurring later within the code needs to be proven. By this, we only capture states in which the assumption holds.[56] We however restrict the scope

[56]Following this approach, a runtime checker, for instance, would immediately stop

of assumptions to be used only in the context of assertions. Invariants and method specifications remain unaffected.

Definition 4.12 (Shortened run)**.** Let $R = \langle s_0, \dots \rangle$ be a run, $M \in \mathcal{M}$ a model field valuator and $\alpha : R \to (\mathbb{N}_+ \nrightarrow A)$ an annotation function.

1. A state $s_i \in R^*$ *misses* its assumptions if there is a $\mathfrak{a} = (\texttt{assume}\ \phi\texttt{;})$ with $\mathfrak{a} \in \alpha^*(s_i)$ such that
$$(s_0, s_i, \lambda_a(0,i), \lambda_w(0,i), M, \upsilon) \not\models \phi$$
where $\upsilon \in \mathcal{U}$ is arbitrary.

2. Let $k \in \mathbb{N}$ be the index position of the first state to miss its assumptions. Then, the sequence $R{\downarrow} := \{s_j \in R^* \mid j < k\}$ is called the *shortened run*. If no state misses its assumptions, we define $R{\downarrow} := R^*$.

Definition 4.13. Let μ be a method with annotated method body π^A and augmented precondition $\overline{pre}$. Let $M \in \mathcal{M}$ be a model field valuator and $s_0 = (h, \sigma, \chi)$ a system state. If μ is non-static with receiver object r let $h(r, \texttt{\created}) = \mathit{true}$ and $\mathit{val}_{s_0}(\texttt{this}) = r$. Let $\mathcal{J}(s_0, \pi^A) = (R, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$ be the black box execution. If $\overline{pre}$ hold in s_0, then it also has to hold:

For every $\mathfrak{a} = (\texttt{assert}\ a\texttt{;}) \in A$ (with a boolean expression) and for every $s_j^* \in R{\downarrow}$ with $\mathfrak{a} \in \alpha^*(s_j^*)$ and for every $\upsilon \in \mathcal{U}$ it is
$$(s_0, s_j^*, \lambda_a(0,j), \lambda_w(0,j), M, \upsilon) \models a$$
μ *respects* its constraints if the above property holds for every finite system state $s \in \mathcal{S}$ and model field valuator $M \in \mathcal{M}$.

Assertions have to be met irregardless of the termination status Ω. They may use the `\old` expression and storage expressions to refer to the pre-state s. Assertions may appear more than once within a run, this is the reason why α is defined as a general relation rather than a function. Consider for example the annotated code in Fig. 4.7. Here, a loop invariant is represented as an annotation within the loop and has to hold on every iteration. The assumption causes every run where the initial value of x is negative not to be regarded. Note that the "old" value of x is the value at the beginning of the enclosing method invocation, not the beginning of the loop.

Historically, JML contained assertion specifications while the Java language did not. Meanwhile, own Java assertions have been introduced [Blo99], which are part of the program and thus not of the specification. If a Java assertion is missed (throughout program execution), an `AssertionError` is thrown. Since this is an instance of `Error`, it is not caught in any way.

processing after reaching an unsatisfied assumption.

```
    public int factorial (int x) {
        int y = 1;
        //@ assume x >= 0;
        while (x != 0) {
            y = y * x--;
            /*@ assert y== (\product int z;
              @          z > x & z <= \old(x); z);
              @*/
        }
        return y;
    }
```

Figure 4.7: Loop invariant as an `assert` statement.

4.3.3 Loop invariants and variants

Invariants and variants are important tools for the specification of loops in programs. In particular, formal verification techniques are able to process programs without loops automatically with ease. In order to verify loops too, invariants are sometimes provided by hand. To have invariants provided directly with the code is very useful this way. And there are even tools like Daikon [EPG+07], which automatically generates possible invariants from from the source code heuristically and puts out JML specifications.[57]

The usefulness of invariants lies in this well-known consequence of mathematical induction: If the invariant holds before the loop is evaluated and it is preserved by every iteration, then it also holds *after* each iteration. Additionally, termination of loops can be proven through the use of *variants* which are non-negative integral expressions which are required to decrease strictly on every iteration. Termination then can be deduced from the fact that there cannot be an infinite decrease.

Compared to other features, the syntax of loop annotations is very simple. They may consist of several invariants indicated by the keyword `maintaining` and several variants indicated by the keyword `decreasing` which are expressions of integral type. The most general form (with B a boolean expression and π the loop body) is given in Fig. 4.8.

As with assertions, there is context required in which loop specifications can make sense. Therefore, we always consider a run of the enclosing method and treat loop specifications similar to assertions. Since the loop condition B might be impure, it is important to which state the loop specification refers.

[57]Another notable work on retrieving invariants by static analysis is [Wei07]. It does not use JML as output language, however.

```
/*@ maintaining Inv_1;
     ⋮
  @ maintaining Inv_n;
  @ decreasing  var_1;
     ⋮
  @ decreasing  var_m;
  @*/
while (B) {
    π
}
```

Figure 4.8: The general appearance a loop specification

In JML, this is the state reached *after* the evaluation of B. Let us assume all invariants and variants to be normalized.

Definition 4.14. Let μ be a method with (receiver o if μ is non-static), annotated body π^A and an augmented precondition $\overline{pre}$. Let $\mathfrak{a}$ be a method specification as shown above. μ *fulfills* the loop specification $\mathfrak{a}$ iff the following holds:

Let $\Sigma = (s_0, s_1, L_a, L_w, \nu, M)$ be a logical state with $\Sigma \models \overline{pre}$ (and if μ is non-static $val_\Sigma(\texttt{this}) = o$). Let the black box execution yield $\mathcal{J}(s_1, \pi) = (R, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$. Let Λ be the (possibly infinite) sequence of states after the evaluation of the loop condition:[58]

$$\Lambda := \{s_i \in R^* \mid \mathfrak{a} \in \alpha^*(s_i)\}$$

Then it holds that:

- The invariant holds for every loop iteration $i \geq 0$:

$$(s_1, s_i, \lambda_a(1,i), \lambda_w(1,i), M, \nu') \models Inv_1 \ \texttt{\&\&} \ \ldots \ \texttt{\&\&} \ Inv_n$$

  where $\nu' \in \mathcal{U}$ is arbitrary

- Every variant strictly decreases on every iteration:
  $0 \leq val_{\vec{\omega}_{j-1}(s_i)}(var_j) < val_{\vec{\omega}_{j-1}(s_{i-1})}(var_j)$ where $\vec{\omega}_k = \omega(var_k)\circ\omega(var_{k-1})\circ \cdots \circ \omega(var_1)$

[58]It might well occur that Λ is still finite while R is not.

As the elements of Λ always denote states at the beginning of the loop, the invariant does not necessarily hold after the loop has been terminated with a `break` statement. If this is desired, it can be added as an assertion.

Variants may have side-effects on each other. Therefore every variant is evaluated in a successor state of the previous evaluation. We assume that invariants and variants do not have side-effects vice versa.

It is vital to evaluate the variant in mathematical arithmetic. Java integers in contrast may cause an infinite loop due to their underflow semantics. The following example does *not* fulfill its specification, even though the invariant is obviously satisfied in every state and the variant is “decreased” through Java’s subtraction operator.

```
    int i;
    //@ maintaining i >= Integer.MIN_VALUE;
    //@ decreasing (i - Integer.MIN_VALUE) /2;
    while (true) {
        i= i - 2;
    }
```

As for `assert` statements, the `\old` expression refers to the pre-state of the method invocation, not the beginning of the loop. Although it is not mentioned in the reference manual, we expect this to apply to frame properties, too, e.g. `\not_assigned` indicates whether the listed location have not been assigned to since the method’s pre-state. In this way, it is not possible to specify frame properties for the loop itself.

Chapter 5

Conclusion

5.1 Related work

This section enumerates works and papers which are related to this work in the sense that their aim is to provide comprehensive semantics for JML. All of them are related to formal verification tools, of which KeY is the only publicly available.

While many features of JML are unsupported by the latter discussed tools, it should be noted that JML is still in development. This means there may be features newly added or, semantics of existing features may have been changed. Notably, the latter case applies to the handling of undefined expressions. This happened in late 2007 and thus postdates everything mentioned in this section.

5.1.1 LOOP

One of the first works on formal semantics for JML is the one by Jacobs and Poll [JP01]. They describe the development of the LOOP tool which is used to verify *Java Card* [Gut97] programs. LOOP translates program code from “essentially all of sequential Java” into theories of Hoare logic describing its semantics. These represent very basic memory operations, rather than abstract method invocations. Reasoning is done through one of the well-known higher order logic tools, PVS [ORR$^+$96] or Isabelle [Pau94]. This treatise so far only covers method specifications, other specification elements are not named.

5.1.2 JIVE

Another notable work has been done by Darvas and Müller [DM07a]. The aim is to provide JML as an input language for the JIVE tool [MMP97]. JIVE is used for interactive verification of programs written in a subset of Java Card, known as Diet Java Card. It is based on the storage logic presented in [Poe97, PM98] and uses Isabelle as back-end prover.

JIVE is targeted at supporting JML *Level 0*. JML language levels have been defined [LPC+08, Sect. 2.9] with the aim to divide the grammar in various levels in order for non-scholars to learn the language more easily.[59] Level 0 is meant to form the core of JML. It does exclude such features as pure method declarations, static invariants and history constraints, loop variants and the `\bigint` type.

Additionally, JIVE does not support any static member of a type, history constraints, annotations or `\TYPE`. Invariants are conjoined to the pre- and postconditions of methods; visible state semantics are not observed. On the other hand, it *does* support (non-static) ghost and model fields as well as data groups.

5.1.3 KeY

KeY [ABHS07, BHS07] is a tool for formal verification of Java Card programs. In contrast to other tools, it does not invoke some back-end higher order logic tool, but uses an own deduction system based on dynamic logic [Bec01]. Originally, KeY only supported OCL as input specification language. Support for JML has been added later, in particular based on Engel's thesis [Eng05, ER07].

Currently, it does not support data groups and `\TYPE`. Assignable clauses are not supported either; it uses a "modifies" clause instead, which is not part of the JML standard. KeY does not use visible state semantics for invariants and history constraints, these are conjoined to pre- and postconditions of methods. On the other hand, ghost and model fields are fully supported. KeY also supports all three math modes (see Sect. 3.3.2), of which `\bigint` is the default. Most notably, KeY is the only tool known to make use of both loop invariants and variants, as well as assignable clauses for loops (which are not part of the JML standard).

[59] This hierarchy is primarily used to hide certain syntactical structures, not to group elements of related semantics. Thus, it is largely unimportant to our work and has been disregarded till now.

5.2 Summary and results

In this thesis, we have developed a rigorous formalization of the Java Modeling Language. Our approach is based on an abstraction of the Java memory model with types and system states. Expressions are evaluated to semantical entities. This respects JML's particularities such as side-effects and a definition of validity which is not based on classical logic. We further sub-divided the expression repertoire into two sub-languages, which evaluation require more or less complexity in semantical underpinning. Finally, we provided semantics for all of JML's specification elements.

Methodologically, we primarily oriented towards formalizing the verbal descriptions of the JML reference manual. This was mostly reliable. On some occasions however, the descriptions are unavailable (e.g. array creation expressions (see Sect. 3.3.7)) or incomplete (e.g. evaluation of locations in frame predicates (see Sect. 3.3.10)). In those cases, we applied our own thoughts, which are mostly backed by Java semantics or the available white papers on JML.

The issue of visible states deserves to be noted separately. Here, we consciously deviated from the official semantics for the reason that those are severely impracticable, respectively they partly contradict the intuitive understanding of invariants. In both cases, this has been discussed with the community (see [Lea09]). Despite those differences, this thesis is the only work on JML semantics so far which extensively comments on the visible state paradigm.

We covered a large variety of JML features. This area coincides with Level 2 of the defined language levels (see [LPC+08, Sect. 2.9]) to a large extent. For instance, our work particularly includes frame predicate expressions or the type `\TYPE`. Since some "advanced" features (in the sense of the language level hierarchy) are purely syntactical sugar, or they are not widely used, we have refrained from naming them. Any feature which occurs in the reference manual, but not in this work, is discussed in Appendix B.

We have provided a formal semantics for JML, which are only based on elementary mathematical notions. We also paid attention on not interfering with Java itself; in our view, JML constitutes as a proper extension to Java. E.g. assertions or ghost variables do not have any effect on computations. In this way, it is applicable of being used as a common ground for the various verification tools.

5.3 Further work

Further work extending this thesis can take on two paths: extending this approach or improving the implementing tools.

There are some features of JML which are left out of focus of this work. These particularly include model types and model methods, which are a topic of on-going research, as well as everything which has to do with concurrency or real-time behavior. The latter however would require rather severe changes in our semantical basis. Although JML provides some facilities for expressing real-time constraints, this not widely used and does not seem to be a primary research topic in the near future. It seems more advantageous to focus on concurrency, which is in fact essential to Java, if not to dedicate oneself to the adaptation to Java 5/6/7 of a forth-coming release of JML.

Although we have tried to be most general and to not focus on particularities of some implementing tool, this thesis has been written with a possible re-write of the JML implementation to the KeY tool in mind. As we have explained in the pre-preceding section, KeY is yet one of the most capable verification tools for JML. In our opinion, there is not much need for sophisticated add-ons. Probable improvements would be an implementation of data groups or support for (static) checking of assertions (although the latter does not necessarily belong to the domain of formal methods).

Appendix A

JML Expression Reference

A.1 Boolean expressions

Definition A.1 (Evaluation of logical operators)**.** Let s be a system state, a and b expressions of type boolean and $\hat{s} := \omega(a)(s)$.

- $val_s(\texttt{!}a) = true$ **iff** $val_s(a) = false$ and $wd_s(a)$
- $val_s(a\ \texttt{|}\ b) = true$ **iff**
  - $val_s(a) = true$ or $val_{\hat{s}}(b) = true$ **and**
  - $wd_s(a)$ and $wd_{\hat{s}}(b)$
- $val_s(a\ \texttt{||}\ b) = true$ **iff**
  - $val_s(a) = true$ and $wd_s(a)$ **or**
  - $wd_s(a)$ and $wd_{\hat{s}}(b)$ and $val_{\hat{s}}(b) = true$
- $val_s(a\ \texttt{\&}\ b) = true$ **iff** $val_s(a) = val_{\hat{s}}(b) = true$ and $wd_s(a)$ and $wd_{\hat{s}}(b)$
- $val_s(a\ \texttt{\&\&}\ b) = true$ **iff** $val_s(a) = val_{\hat{s}}(b) = true$ and $wd_s(a)$ and $wd_{\hat{s}}(b)$
- $val_s(a\ \texttt{==>}\ b) = true$ **iff**
  - $val_s(a) = false$ and $wd_s(a)$ **or**
  - $wd_s(a)$ and $wd_{\hat{s}}(b)$ and $val_{\hat{s}}(b) = true$
- $val_s(a\ \texttt{<==}\ b) = true$ **iff**
  - $val_s(a) = true$ and $wd_s(a)$ **or**
  - $wd_s(a)$ and $wd_{\hat{s}}(b)$ and $val_{\hat{s}}(b) = false$

- $val_s(a \texttt{ <==> } b) = true$ **iff** $val_s(a) = val_{\hat{s}}(b)$ and $wd_s(a)$ and $wd_{\hat{s}}(b)$
- $val_s(a\texttt{\^{}}b) = true$ **iff** $val_s(a) = val_{\hat{s}}(b)$ and $wd_s(a)$ and $wd_{\hat{s}}(b)$
- $val_s(a \texttt{ <!=> } b) = true$ **iff** $val_s(a) \neq val_{\hat{s}}(b)$ and $wd_s(a)$ and $wd_{\hat{s}}(b)$

Definition A.2 (Well-definition of logical operators)**.** Let s be a system state, a and b expressions of type boolean and $\hat{s} := \omega(a)(s)$.

- $wd_s(\texttt{!}a)$ **iff** $wd_s(a)$
- $wd_s(a \texttt{ || } b)$ and $wd_s(a \texttt{ <== } b)$ **iff**
  - $wd_s(a)$ and $val_s(a) = true$ **or**
  - $wd_s(a)$ and $wd_{\hat{s}}(b)$
- $wd_s(a \texttt{ \&\& } b)$ and $wd_s(a \texttt{ ==> } b)$ **iff**
  - $wd_s(a)$ and $val_s(a) = false$ **or**
  - $wd_s(a)$ and $wd_{\hat{s}}(b)$
- $wd_s(a \texttt{ | } b)$, $wd_s(a \texttt{ \& } b)$, $wd_s(a \texttt{ \^{} } b)$, $wd_s(a \texttt{ <==> } b)$ and $wd_s(a \texttt{ <!=> } b)$ **iff** $wd_s(a)$ and $wd_{\hat{s}}(b)$

Definition A.3 (State-transition function for logical operators)**.** Let s be a system state and a and b expressions of type boolean.

- $\omega(\texttt{!}a) = \omega(a)$
- $\omega(a \texttt{ | } b) = \omega(a \,\&\, b) = \omega(a \texttt{ \^{} } b) = \omega(a \texttt{ <==> } b) = \omega(a \texttt{ <!=> } b) = \omega(b) \circ \omega(a)$
- $\omega(a \texttt{ || } b)(s) = \omega(a \texttt{ <== } b)(s) = \begin{cases} \omega(a)(s) & val_s(a) = true \\ \omega(b)(\omega(a)(s)) & val_s(a) = false \end{cases}$
- $\omega(a \texttt{ \&\& } b)(s) = \omega(a \texttt{ ==> } b)(s) = \begin{cases} \omega(b)(\omega(a)(s)) & val_s(a) = true \\ \omega(a) & val_s(a) = false \end{cases}$

Definition A.4 (Equality predicate operators)**.** Let a and b be expressions of compatible types T_1 and T_2. Let n and m be expressions of numerical types. Let s be a system state and $\hat{s} := \omega(a)(s)$ or $\omega(n)(s)$ respectively.

- $val_s(a \texttt{ == } b) = true$ **iff** $val_s(a) = val_{\hat{s}}(b)$ and $val_s(a) \neq NaN$
- $val_s(a \texttt{ != } b) = true$ **iff** $val_s(a \texttt{ == } b) = false$

- $val_s(n \texttt{ < } m) = true$ **iff** $val_s(n) < val_{\hat{s}}(m)$ and $val_s(n), val_{\hat{s}}(m) \neq NaN$
- $val_s(n \texttt{ <= } m) = true$ **iff** $val_s(n \texttt{ < } m) = true$ or $val_s(n \texttt{ == } m) = true$
- $val_s(n \texttt{ > } m) = true$ **iff** $val_s(n) > val_{\hat{s}}(m)$ and $val_s(n), val_{\hat{s}}(m) \neq NaN$
- $val_s(n \texttt{ => } m) = true$ **iff** $val_s(n \texttt{ > } m) = true$ or $val_s(n \texttt{ == } m) = true$

For every equality predicate $(a \star b)$ it is $wd_s(a \star b)$ **iff** $wd_s(a)$ and $wd_{\hat{s}}(b)$. And $\omega(a \star b) = \omega(b) \circ \omega(a)$.

A.2 Other simple expressions

Definition A.5 (Numerical operators)**.** Let n, m be expressions of exact dynamic integer types T_1 and T_2, $\star \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{<<}, \texttt{>>}, \texttt{>>>}, \texttt{\&}, \texttt{|}, \texttt{\^{}}\}$ (binary) and $\diamond \in \{\texttt{+}, \texttt{-}, \texttt{\~{}}\}$ (unary). Let s be a system state and $\hat{s} := \omega(n)(s)$.

- $val_s(n \star m) = \begin{cases} \text{arbitrary} & \star \in \{\texttt{/}, \texttt{\%}\} \text{ and } val_{\hat{s}}(m) = 0 \\ val_s(n) \star_{T'} val_{\hat{s}}(m) & \text{otherwise} \end{cases}$
  where $\star_{T'}$ represents the corresponding mathematical operation with the modulo/overflow semantics of type $T' := \begin{cases} \text{long} & T_1 \text{ or } T_2 = \text{long} \\ \text{int} & \text{otherwise} \end{cases}$.
- $wd_s(n \star m)$ **iff**
  - $wd_s(n)$ and $wd_{\hat{s}}(m)$ **and**
  - $\star \notin \{\texttt{/}, \texttt{\%}\}$ or $val_{\hat{s}}(m) \neq 0$
- $\omega(n \star m) = \omega(m) \circ \omega(n)$
- $val_s(\diamond n) = \diamond_{T_1} val_s(n)$
- $wd_s(\diamond n)$ **iff** $wd_s(n)$
- $\omega(\diamond n) = \omega(n)$

Let e, f be expressions of exact dynamic floating point types T_1 and T_2 (WLOG $T_1 \sqsubseteq T_2$) and $\star \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$, $\pm \in \{\texttt{+}, \texttt{-}\}$. Let s be a system state and $\hat{s} := \omega(e)(s)$.

- $val_s(e \star f) = val_s(e) \star_{T_2} val_{\hat{s}}(f)$
- $wd_s(e \star f)$ **iff** $wd_s(e)$ and $wd_{\hat{s}}(f)$
- $\omega(e \star f) = \omega(f) \circ \omega(e)$

- $val_s(\pm e) = \pm_{T_2} val_s(e)$
- $wd_s(\pm e)$ **iff** $wd_s(e)$
- $\omega(\pm e) = \omega(e)$

Let x, y be expressions of exact dynamic type `\bigint` or `\real` and $\star \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$, $\pm \in \{\texttt{+}, \texttt{-}\}$. Let s be a system state and $\hat{s} := \omega(x)(s)$.

- $val_s(x \star y) = \begin{cases} \text{arbitrary} & \star \in \{\texttt{/}, \texttt{\%}\} \text{ and } val_{\hat{s}}(y) = 0 \\ val_s(x) \star_{\mathbb{R}} val_{\hat{s}}(y) & \text{otherwise} \end{cases}$
  where $\star_{\mathbb{R}}$ represents the corresponding mathematical operation
- $wd_s(x \star y)$ **iff**
  - $wd_s(x)$ and $wd_{\hat{s}}(y)$ **and**
  - $\star \notin \{\texttt{/}, \texttt{\%}\}$ or $val_{\hat{s}}(y) \neq 0$
- $\omega(x \star y) = \omega(y) \circ \omega(x)$
- $val_s(\pm x) = \pm val_s(x)$
- $wd_s(\pm x)$ **iff** $wd_s(x)$
- $\omega(\pm x) = \omega(x)$

Definition A.6 (Conditional expression)**.** Let b be a boolean expression and c, d expressions of comparable types. Let s be a system state and $\hat{s} := \omega(b)(s)$.

- $val_s(b?c\!:\!d) = \begin{cases} val_{\hat{s}}(c) & val_s(b) = true \\ val_{\hat{s}}(d) & val_s(b) = false \end{cases}$
- $wd(b?c\!:\!d)$ **iff**
  - $val_s(b) = true$ and $wd_s(b)$ and $wd_{\hat{s}}(c)$ **or**
  - $val_s(b) = false$ and $wd_s(b)$ and $wd_{\hat{s}}(d)$
- $\omega(b?c\!:\!d)(s) = \begin{cases} \omega(c)(\hat{s}) & val_s(b) = true \\ \omega(d)(\hat{s}) & val_s(b) = false \end{cases}$

Definition A.7 (Type expressions)**.** Let T be a type, e be an expression and and t an expression of type `\TYPE`.

- $val(\texttt{\textbackslash type}(T)) = T$
- $wd(\texttt{\textbackslash type}(T))$ **iff** $T \in V_{\texttt{\textbackslash TYPE}}$

- $\omega(\texttt{\textbackslash type}(T)) = id$
- $$val(\texttt{\textbackslash typeof}(e)) = \begin{cases} \texttt{Boolean} & val(e) \in V_{\texttt{bool}} \\ \texttt{Character} & val(e) \in V_{\texttt{char}} \\ \texttt{Byte} & val(e) \in V_{\texttt{byte}} \\ \texttt{Short} & val(e) \in V_{\texttt{short}} \setminus V_{\texttt{byte}} \\ \texttt{Integer} & val(e) \in V_{\texttt{int}} \setminus V_{\texttt{short}} \\ \texttt{Long} & val(e) \in V_{\texttt{long}} \setminus V_{\texttt{int}} \\ \texttt{Float} & val(e) \in V_{\texttt{float}} \setminus V_{\texttt{long}} \\ \texttt{Double} & val(e) \in V_{\texttt{double}} \setminus V_{\texttt{float}} \\ \tilde{T} & val(e) \in V^0_{\tilde{T}} \text{ where } \tilde{T} \sqsubseteq \texttt{Object} \\ \text{arbitrary} & \text{otherwise} \end{cases}$$
- $wd(\texttt{\textbackslash typeof}(e))$ **iff** $val(e) \in \bigcup_{T \in V_{\texttt{\textbackslash TYPE}}} V^0_T$
- $\omega(\texttt{\textbackslash typeof}(e)) = \omega(e)$
- $$val(\texttt{\textbackslash elemtype}(t)) = \begin{cases} T' & val(t) = T'\texttt{[]} \\ \text{null} & \text{otherwise} \end{cases}$$
- $wd(\texttt{\textbackslash elemtype}(t))$ **iff** $wd(t)$
- $\omega(\texttt{\textbackslash elemtype}(t)) = \omega(t)$

Definition A.8 (Type cast)**.** Let T' be a type and a an expression of type T.

- $$val(\texttt{(}T'\texttt{)}\ a) = \begin{cases} cast(T, T', val(a)) & T \text{ and } T' \text{ numerical} \\ val(a) & \text{otherwise} \end{cases}$$
- $wd(\texttt{(}T'\texttt{)}\ a)$ **iff** $val(\texttt{(}T'\texttt{)}\ a) \in V_{T'}$
- $\omega(\texttt{(}T'\texttt{)}\ a) = \omega(a)$

Where the numerical promotion function $cast : \mathcal{T} \times \mathcal{T} \times \mathcal{U} \to \mathcal{U}$ is given as follows: (Where rtn_M rounds to the nearest element of M and rtz rounds to the next integer towards zero.)

- $cast(I, F, a) = \text{rtn}_{V_T}(a)$ for I integral, F floating point type
- $cast(I_x, I_y, a) = ls(rs(a) \mod 2^y)$ for I_n are n-bit signed integral types where $rs(b) = \begin{cases} b & b \geq 0 \\ b + 2^x & b < 0 \end{cases}$ and $ls(b) = \begin{cases} b - 2^y & b \geq 2^{y-1} \\ b & b < 2^{y-1} \end{cases}$
- $cast(I_x, \text{char}, a) = rs(a) \mod 2^{16}$
  where V_{char} is identified with $\{0, \ldots, 2^{16} - 1\}$

- $cast(\text{char}, I_y, a) = ls(a \mod 2^y)$
- $cast(F, L, a) = \begin{cases} 0 & a = NaN \\ \max(\min V_L, \text{rtz}(a)) & a \leq 0 \\ \min(\max V_L, \text{rtz}(a)) & a > 0 \end{cases}$
  for F floating point type, $L \in \{\text{int}, \text{long}\}$
- $cast(F, S, a) = cast(S, \text{int}, cast(\text{int}, F, a))$ for F floating point type, $S \in \{\text{short}, \text{byte}, \text{char}\}$
- $cast(T, F, a) = \begin{cases} +\infty & a > \max \mathbb{Q}_F \\ -\infty & a < \min \mathbb{Q}_F \\ \text{rtn}_{V_F} & \text{otherwise} \end{cases}$
  for $T \in \{\texttt{\bigint}, \texttt{\real}\}$, F floating point type
- $cast(T, I, a) = cast(\text{double}, I, cast(T, \text{double}, a))$
  for $T \in \{\texttt{\bigint}, \texttt{\real}\}$, I integral type
- $cast(F, \texttt{\real}, a) = a$ for F floating point type
- $cast(F, \texttt{\bigint}, a) = cast(\texttt{\real}, \texttt{\bigint}, cast(F, \texttt{\real}, a))$ for F floating point type
- $cast(\texttt{\real}, \texttt{\bigint}, a) = \text{rtn}_{\mathbb{Z}}(a)$
- $cast(T_1, T_2, a) = a$ otherwise

A.3 Reference expressions

Definition A.9 (Values of reference expressions)**.** Let $s = (h, \sigma, \chi)$ be a system state. Let $\Sigma = (s_0, s_1, L_a, L_w, \rho, M)$ be a logical state.

- $val_s(c) = c$ for a constant (primitive value) c
- $val_s(v) = \begin{cases} \sigma(v) & v \in \text{dom}(\sigma) \\ \text{arbitrary} & \text{otherwise} \end{cases}$ for a local variable v
- $val_s(\texttt{this}) = \begin{cases} o & (\mu, o) = \text{top}(\chi) \text{ for some method } \mu \\ \text{arbitrary} & \text{otherwise} \end{cases}$
- $val_\Sigma(\texttt{\result}) = \rho$
- $val_s(r.x) = \hat{h}(val_s(r), x)$ for an expression r of reference type and x a non-static non-model field identifier with $\omega(a)(s) = (\hat{h}, \dots)$

- $val_\Sigma(r.x) = \begin{cases} M_{\omega(r)(s_1)}(val_{s_1}(r), x) & M \neq \emptyset \\ \text{arbitrary} & M = \emptyset \end{cases}$ for an expression r of reference type and x a non-static model field identifier

- $val_s(T.x) = h(o, x)$ for a static non-model field x defined in type T with $o \in V_T$

- $val_\Sigma(T.x) = \begin{cases} M_{s_1}(o, x) & M \neq \emptyset \\ \text{arbitrary} & M = \emptyset \end{cases}$ for a static model field x defined in type T with $o \in V_T$

- $val_s(a\texttt{[}n\texttt{]}) = \check{h}(val_s(a), val_{\omega(a)(s)}(n))$ for an expression a of array type and n expression of an integer type with $\omega(n)(\omega(a)(s)) = (\check{h}, \dots)$

Definition A.10 (Location expressions)**.**

- $loc_s(r.x) = \{(val_s(r), x)\}$ for an expression r of reference type and x a (possibly model) field identifier

- $loc_s(T.x) = \{(o, x) \mid o \in V_C\}$ for a static (possibly model) field identifier x in type T

- $loc_s(r.*) = \{(val_s(r), \iota) \mid \iota \in \mathbb{I}\} \cap \mathcal{L}'$

- $loc_s(T.*) = \{(o, \iota) \mid o \in V_T, \iota \in \mathbb{I}\} \cap \mathcal{L}'$

- $loc_s(a\texttt{[}n\texttt{]}) = \{(val_s(a), val_{\omega(a)(s)}(n))\}$ for an expression a of array type and n expression of an integer type

- $loc_s(a\texttt{[}n..m\texttt{]}) = \{(val_s(a), k) \mid val_{\omega(a)(s)}(n) \leq k \leq val_{\omega(a)(s)}(m)\}$ for an expression a of array type and n, m expressions of integer types

- $loc_s(a\texttt{[*]}) = loc_s(a\texttt{[}0 \;..\; val_s(a.\texttt{length}) - 1\texttt{]})$

- $loc_s(e_1, e_2, \dots, e_n) = \bigcup_{i=1}^n loc_{\vec{\omega}_{i-1}(s)}(e_i)$
  with $\vec{\omega}_i := \omega(e_i) \circ \omega(e_{i-1}) \circ \dots \circ \omega(e_1)$ and $\vec{\omega}_0 := id$ for a list of references

- $loc(\texttt{\nothing}) = \emptyset$

- $loc(\texttt{\everything}) = \mathcal{L}'$

Definition A.11 (Well-definition of reference expressions)**.** Let everything be as above.

- $wd(c)$, $wd(\texttt{\result})$, $wd(T.*)$, $wd(\texttt{\nothing})$, $wd(\texttt{\everything})$ always hold

- $wd_s(v)$ **iff** $v \in \mathrm{dom}(\sigma)$
- $wd_s(\texttt{this})$ **iff** $\mathrm{top}(\chi) \in M \times \mathcal{U}$
- $wd_s(r.\texttt{\textbackslash created})$ **iff** $wd_s(r)$ and $val_s(r) \neq null$
- $wd_s(r.x)$ with $x \neq \texttt{\textbackslash created}$ **iff**
  - $wd_s(r)$,
  - $val_s(r) \neq null$ and
  - $val_s(r.\texttt{\textbackslash created}) = true$
  - and if x is a model field, then also $M \neq \emptyset$
- $wd_s(r.{*})$ **iff** $wd_s(r)$, $val_s(r) \neq null$ and $val_s(r.\texttt{\textbackslash created}) = true$
- $wd_s(T.x)$ **iff** x is concrete or $M \neq \emptyset$
- $wd_s(a\texttt{[}n\texttt{]})$ **iff** $wd_s(a)$, $val_s(a) \neq null$, $val_s(a.\texttt{\textbackslash created}) = true$, $wd_{\omega(a)(s)}(n)$ and $0 \leq val_{\omega(a)(s)}(n) < val_s(a.\texttt{length})$
- $wd_s(a\texttt{[}n..m\texttt{]})$ **iff** $wd_s(a)$, $val_s(a) \neq null$, $val_s(a.\texttt{\textbackslash created}) = true$ $wd_{\omega(a)(s)}(n)$, $wd_{\omega(a)(\omega(n)(s))}(m)$ and $0 \leq val_{\omega(a)(s)}(n) \leq val_{\omega(a)(\omega(n)(s))}(m) < val_s(a.\texttt{length})$
- $wd_s(a\texttt{[*]})$ **iff** $wd_s(a)$ and $val_s(a) \neq null$ and $val_s(a.\texttt{\textbackslash created}) = true$
- $wd_s(e_1, e_2, \ldots, e_n)$ **iff** $wd_{\vec{\omega}_{i-1}(s)}(e_i)$ for every $i \in [1, n]$

Definition A.12 (State transition function for reference expressions)**.** Let everything be as above.

- $\omega(c) = \omega(v) = \omega(\texttt{this}) = \omega(\texttt{\textbackslash result}) = \omega(T.{*}) = \omega(\texttt{\textbackslash nothing}) = \omega(\texttt{\textbackslash everything}) = id$
- $\omega(r.x) = \begin{cases} \omega(\psi) \circ \omega(r) & x \text{ is a model field with } (val(r), x, \psi) \in \mathfrak{M} \\ \omega(r) & \text{otherwise} \end{cases}$
- $\omega(r.{*}) = \omega(r)$
- $\omega(T.x) = \begin{cases} \omega(\psi) & x \text{ is a model field with } (o, x, \psi) \in \mathfrak{M} \text{ for } o \in V_T \\ id & \text{otherwise} \end{cases}$
- $\omega(a\texttt{[}n\texttt{]}) = \omega(n) \circ \omega(a)$
- $\omega(a\texttt{[}n..m\texttt{]}) = \omega(m) \circ \omega(n) \circ \omega(a)$

- $\omega(a\texttt{[*]}) = \omega(a)$
- $\omega(e_1, e_2, \ldots, e_n) = \vec{\omega}_n$

Definition A.13 (Pure method invocation).

- Let $s = (h, \sigma, \chi)$ be a system state, a an expression of reference type, $e_1, \ldots, e_n$ expressions of respective types $T_1, \ldots, T_n$ and

  $$\mathrm{msp}(m, \langle T_1, \ldots, T_n\rangle, val_s(a)) = \mu = (m, \pi, \langle (T_1, \iota_1), \ldots, (T_n, \iota_n)\rangle, T_R)$$

  a pure (non-void) non-static method.

  - $val_s(a.m(e_1, \ldots, e_n)) = \rho \in V_{T_R}$
  - $wd_s(a.m(e_1, \ldots, e_n))$ **iff** $wd_s(a)$, $val_s(a) \neq \mathrm{null}$, $val_s(a.\texttt{\created}) = true$ and $wd_{\vec{\omega}_{i-1}}(e_i)$ and $\Omega = \mathrm{null}$
  - $\omega(a.m(e_1, \ldots, e_n))(s) = (h', \sigma, \chi)$

  Where

  - $\mathcal{J}(\tilde{s}, \pi) = (\langle s_0, \ldots, s_k\rangle, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$ with $s_k = (h', \sigma', \chi')$,
  - $\tilde{s} = \vec{\omega}_n((h, \sigma^{\{\iota_j \mapsto val_{\vec{\omega}_{j-1}(s)}(e_j) | 1 \leq j \leq n\}}, \mathrm{push}(\chi, (\mu, val_s(a)))))$,
  - $\vec{\omega}_j := \omega(e_j) \circ \omega(e_{j-1}) \circ \ldots \circ \omega(e_1) \circ \omega(a)$.

- Let $s = (h, \sigma, \chi)$ be a system state, $e_1, \ldots, e_n$ expressions of respective types $T_1, \ldots, T_n$ and $\mu = (m, \pi, \langle (T_1, \iota_1), \ldots, (T_n, \iota_n)\rangle, T_R)$ a pure (non-void non-constructor) static method of class C

  - $val_s(C.m(e_1, \ldots, e_n)) = \rho \in V_{T_R}$
  - $wd_s(C.m(e_1, \ldots, e_n))$ **iff** $val_s(t) \neq \mathrm{null}$, $wd_{\vec{\omega}_{i-1}(s)}(e_i)$ and $\Omega = \mathrm{null}$
  - $\omega(C.m(e_1, \ldots, e_n))(s) = (h', \sigma, \chi)$

  Where

  - $\mathcal{J}(\tilde{s}, \pi) = (\langle s_0, \ldots, s_k\rangle, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$ with $s_k = (h', \sigma', \chi')$,
  - $\tilde{s} = \vec{\omega}_n((h, \sigma^{\{\iota_j \mapsto val_{\vec{\omega}_{j-1}(s)}(e_j) | 1 \leq j \leq n\}}, \mathrm{push}(\chi, \mu)))$,
  - $\vec{\omega}_j := \omega(e_j) \circ \omega(e_{j-1}) \circ \ldots \circ \omega(e_1)$, $\vec{\omega}_0 := id$.

- Let $s = (h, \sigma, \chi)$ be a system state, $e_1, \ldots, e_n$ expressions of respective types $T_1, \ldots, T_n$ and $\mu = (T, \pi, \langle (T_1, \iota_1), \ldots, (T_n, \iota_n)\rangle, T)$ a pure constructor of type T[60]

[60]This might be the (implicitly given) default constructor if none is explicitly defined.

- $val_s(\texttt{new } T(e_1, \ldots, e_n)) = \rho \in V_T$
- $wd_s(\texttt{new } T(e_1, \ldots, e_n))$ **iff** $wd_{\vec{\omega}_{i-1}(s)}(e_i)$ and $\Omega = \text{null}$
- $\omega(\texttt{new } T(e_1, \ldots, e_n))(s) = (h', \sigma, \chi)$

Where

- $\rho \in V_T^0$ is a fresh domain element,
- $\mathcal{J}(\tilde{s}, \pi) = (\langle s_0, \ldots, s_k \rangle\rangle, \lambda_a, \lambda_w, \alpha, \Omega, \rho)$ with $s_k = (h', \sigma', \chi')$,
- $\tilde{s} = \vec{\omega}_n((\tilde{h}, \sigma^{\{\iota_j \mapsto val_{\vec{\omega}_{j-1}(s)}(e_j) | 1 \leq j \leq n\}}, \text{push}(\chi, (\mu, \rho))))$
- $\tilde{h}$ coincides with h except for $\tilde{h}(\rho, \texttt{\textbackslash created}) = true$ and for every non-static field x in T it is $\tilde{h}(\rho, x) = d(T(x))$,
- $\vec{\omega}_j := \omega(e_j) \circ \omega(e_{j-1}) \circ \ldots \circ \omega(e_1)$, $\vec{\omega}_0 := id$.

Definition A.14 (New array declaration)**.** Let s be a system state.

- (Array initializer) Let $e_0, \ldots, e_{n-1}$ be expressions of respective (comparable) types $T_0, \ldots, T_{n-1}$. Let T be a least common super-type. Let $\vec{\omega}_j := \omega(e_j) \circ \omega(e_{j-1}) \circ \cdots \circ \omega(e_0)$, $\vec{\omega}_{-1} := id$ and $\vec{\omega}_{n-1}(s) = (h', \sigma', \chi')$.
  - $val_s(\texttt{\{}e_0, \ldots, e_{n-1}\texttt{\}}) = a$ where $a \in V^0_{T\texttt{[]}}$ is a fresh domain element
  - $wd_s(\texttt{\{}e_0, \ldots, e_{n-1}\texttt{\}})$ **iff** $wd_{\vec{\omega}_{j-1}(s)}(e_j)$ for every $j \in [0, n-1]$
  - $\omega(\texttt{\{}e_0, \ldots, e_{n-1}\texttt{\}})(s) = (h'', \sigma', \chi')$ where h'' coincides with h' except for
    * $h''(a, \texttt{\textbackslash created}) = true$
    * $h''(a, \texttt{length}) = n$
    * $h''(a, i) = val_{\vec{\omega}_{i-1}}(e_i)$ for every $i \in [0, n-1]$

- (`new` array invocation) Let T be a type, $n_1, \ldots, n_k$ expressions of type `int` and $\underline{init}$ an array initializer. Let $\vec{\omega}_j := \omega(n_j) \circ \omega(n_{j-1}) \circ \cdots \circ \omega(n_1)$. Let $T\texttt{[]}^k$ denote the result type $T\underbrace{\texttt{[]} \ldots \texttt{[]}}_{k}$.
  - $val_s(\texttt{new } T\texttt{[}n_1\texttt{]}\texttt{[}n_2\texttt{]} \cdots \texttt{[}n_k\texttt{]}\underline{init}) = val_{\vec{\omega}_k(s)}(\underline{init})$
  - $wd_s(\texttt{new } T\texttt{[}n_1\texttt{]}\texttt{[}n_2\texttt{]} \cdots \texttt{[}n_k\texttt{]}\underline{init})$ **iff** $val_{\vec{\omega}_{i-1}(s)}(n_i) \geq 0$ for every $i \in [1, k]$, $wd_{\vec{\omega}_k(s)}(\underline{init})$ and $val_{\vec{\omega}_k(s)}(\underline{init}) \in V_{T\texttt{[]}^k}$
  - $\omega(\texttt{new } T\texttt{[}n_1\texttt{]}\texttt{[}n_2\texttt{]} \cdots \texttt{[}n_k\texttt{]}\underline{init}) = \omega(\underline{init}) \circ \vec{\omega}_k$

Definition A.15 (`\old`)**.** Let $s_0, s_1 \in \mathcal{S}$ be system states and *expr* some expression.

- $val_{(s_0,s_1)}(\texttt{\textbackslash old}(expr)) = val_{(s_0,s_0)}(expr)$
- $wd_{(s_0,s_1)}(\texttt{\textbackslash old}(expr))$ **iff** $wd_{(s_0,s_0)}(expr)$ and
- $\omega(\texttt{\textbackslash old}(expr)) = id$.

A.4 JML predicates

Definition A.16 (Type predicates)**.** Let t_1, t_2 be expressions of type \TYPE and e an expression of type T.

- $val_s(t_1 \texttt{ <: } t_2) = true$ **iff** $val_s(t_1) \sqsubseteq val_{\omega(t_1)(s)}(t_2)$ or $val_s(t_1) = val_{\omega(t_1)(s)}(t_2) \in \{\text{null}, \texttt{void}\}$
- $wd_s(t_1 \texttt{ <: } t_2)$ **iff** $wd_s(t_1)$ and $wd_{\omega(t_1)(s)}(t_2)$
- $\omega(t_1 \texttt{ <: } t_2) = \omega(t_2) \circ \omega(t_1)$
- $val_s(e \texttt{ instanceof } t_2) = true$ **iff** $T' \sqsubseteq val_{\omega(e)(s)}(t_2)$ with $val_s(e) \in V^0_{T'}$
- $wd_s(e \texttt{ instanceof } t_2)$ **iff** $wd_s(e)$, $wd_{\omega(e)(s)}(t_2)$, and $val_s(e) \neq null$
- $\omega(e \texttt{ instanceof } t_2) = \omega(t_2) \circ \omega(e)$

Definition A.17 (Frame predicates)**.** Let $\Sigma = (s_0, s_1, L_a, L_w, \rho, M)$ be a logical state with $s_j = (h_j, \sigma_j, \chi_j)$. Let Λ be a list of normalized storage reference expressions, $L_\Lambda := loc_{s_0}(\Lambda)$ and $L^* = \{(o, x) \in \mathcal{L} \mid h_0(o, \texttt{\textbackslash created}) = false\}$.

- $val_\Sigma(\texttt{\textbackslash not\_assigned}(\Lambda)) = true$ **iff** $L_w \cap \bigcup_{\ell \in L_\Lambda} \mathcal{D}(\ell) \subseteq L^*$
- $val_\Sigma(\texttt{\textbackslash only\_accessed}(\Lambda)) = true$ **iff** $L_a \setminus \bigcup_{\ell \in L_\Lambda} \mathcal{D}(\ell) \subseteq L^*$
- $val_\Sigma(\texttt{\textbackslash only\_assigned}(\Lambda)) = true$ **iff** $L_w \setminus \bigcup_{\ell \in L_\Lambda} \mathcal{D}(\ell) \subseteq L^*$
- $val_\Sigma(\texttt{\textbackslash not\_modified}(\Lambda)) = true$ **iff** for every $\ell' \in \bigcup_{\ell \in L_\Lambda} \mathcal{D}(\ell)$ it is $h_0(\ell') = h_1(\ell')$

For any frame predicate $\mathfrak{F}$ it is $wd_\Sigma(\mathfrak{F}(\Lambda))$ iff $wd_{s_0}(\Lambda)$. And $\omega(\mathfrak{F}(\Lambda)) = \omega(\Lambda)$.

Definition A.18 ($\texttt{\textbackslash fresh}$)**.** Let $e_1, \ldots, e_n$ be expressions of reference types and $s_0 = (h_0, \sigma_0, \chi_0)$, $s_1 = (h_1, \sigma_1, \chi_1)$ be system states.
$val_{(s_0,s_1)}(\texttt{\textbackslash fresh}(e_1, \ldots, e_n)) = true$ **iff** for every $i \in [1, n]$

- $val_{\vec{\omega}_{i-1}(s_1)}(e_i) \neq \text{null}$,
- $val_{\vec{\omega}_{i-1}(s_1)}(e_i.\texttt{\textbackslash created}) = \mathit{true}$ **and**
- $h_0(val_{\vec{\omega}_{i-1}(s_1)}(e_i), \texttt{\textbackslash created}) = \mathit{false}$

where $\vec{\omega}_k := \omega(e_k) \circ \omega(e_{k-1}) \circ \ldots \circ \omega(e_1)$ and $\vec{\omega}_0 := \mathit{id}$.

- $wd_{(s_0,s_1)}(\texttt{\textbackslash fresh}(e_1, \ldots, e_n)))$ **iff** $wd_{\vec{\omega}_{i-1}(s_1)}(e_i)$ for every $i \in [1, n]$
- $\omega(\texttt{\textbackslash fresh}(e_1, \ldots, e_n)) = \vec{\omega}_n$

Definition A.19 (`\nonnullelements`)**.** Let a be an expression of array type and s a system state.

- $val_s(\texttt{\textbackslash nonnullelements}(a)) = \mathit{true}$ **iff**
  - $val_s(a) \neq \text{null}$ and
  - for every $i \in \{0, \ldots, val_s(a.\texttt{length})\} : val_s(a\texttt{[}i\texttt{]}) \neq \text{null}$
- It is $wd_s(\texttt{\textbackslash nonnullelements}(a))$ **iff** $wd_s(a)$ and $val_s(a) \in V_{T\texttt{[]}}$ for some type $T \in \mathcal{T}$
- $\omega(\texttt{\textbackslash nonnullelements}(a)) = \omega(a)$.

A.5 Quantifiers

Definition A.20 (Logical quantifiers)**.** Let T be a type, x an identifier and a, b boolean expressions.

- $val_{(s_0,s_1)}(\texttt{\textbackslash forall nullable } T\ x; a; b) = \mathit{true}$ **iff** for all $o \in V_T$ it is

$$val_{(s_0^{\{x\mapsto o\}}, s_1^{\{x\mapsto o\}})}(a \texttt{ ==> } b) = \mathit{true}$$

- $val_{(s_0,s_1)}(\texttt{\textbackslash exists nullable } T\ x; a; b) = \mathit{true}$ **iff** for some $o \in V_T$ it is

$$val_{(s_0^{\{x\mapsto o\}}, s_1^{\{x\mapsto o\}})}(a \texttt{ \&\& } b) = \mathit{true}$$

- $wd_{(s_0,s_1)}(\texttt{\textbackslash forall nullable } T\ x; a; b)$ **iff** for all $o \in V_T$ it is

$$wd_{(s_0^{\{x\mapsto o\}}, s_1^{\{x\mapsto o\}})}(a \texttt{ ==> } b)$$

- $wd_{(s_0,s_1)}(\texttt{\textbackslash exists nullable}\ T\ x; a; b)$ **iff** for some $o \in V_T$ it is

$$wd_{(s_0^{\{x\mapsto o\}},s_1^{\{x\mapsto o\}})}(a\ \texttt{\&\&}\ b)$$

For both quantifiers it is $\omega(\ldots) = id$.

Definition A.21 (Generalized quantifiers)**.** Let T be a type, x an identifier, b a boolean expression and e an expression of numerical type T'. Let $Z := \{z \in V_T \mid val_{s^{\{x\mapsto z\}}}(b) = true\}$.

- $val_s(\texttt{\textbackslash sum nullable}\ T\ x; b; e) =$

$$\begin{cases} \text{arbitrary} & |Z| = \infty \\ \bigoplus\limits_{z\in V_T} val_{s^{\{x\mapsto z\}}}(b\ ?\ e:\ 0) & |Z| < \infty, T' \text{ integral} \\ cast(\texttt{\textbackslash real}, T', \sum\limits_{z\in V_T} val_{s^{\{x\mapsto z\}}}(b\ ?\ e:\ 0)) & \text{otherwise} \end{cases}$$

- $val_s(\texttt{\textbackslash product nullable}\ T\ x; b; e) =$

$$\begin{cases} \text{arbitrary} & |Z| = \infty \\ \bigotimes\limits_{z\in V_T} val_{s^{\{x\mapsto z\}}}(b\ ?\ e:\ 1) & |Z| < \infty, T' \text{ integral} \\ cast(\texttt{\textbackslash real}, T', \prod\limits_{z\in V_T} val_{s^{\{x\mapsto z\}}}(b\ ?\ e:\ 1)) & \text{otherwise} \end{cases}$$

- $val_s(\texttt{\textbackslash max nullable}\ T\ x; b; e) =$

$$\begin{cases} \text{arbitrary} & |Z| = \infty \\ \text{arbitrary} & Z = \emptyset, T' \in \{\texttt{\textbackslash bigint}, \texttt{\textbackslash real}\} \\ \min V_{T'} & Z = \emptyset, T' \notin \{\texttt{\textbackslash bigint}, \texttt{\textbackslash real}\} \\ \max \tilde{Z} & \text{otherwise} \end{cases}$$

  where $\tilde{Z} := \{val_{\omega(b)(s^{\{x\mapsto z\}})}(e) \mid z \in Z\}$.

- $val_s(\texttt{\textbackslash min nullable}\ T\ x; b; e) =$

$$\begin{cases} \text{arbitrary} & |Z| = \infty \\ \text{arbitrary} & Z = \emptyset, T' \in \{\texttt{\textbackslash bigint}, \texttt{\textbackslash real}\} \\ \max V_{T'} & Z = \emptyset, T' \notin \{\texttt{\textbackslash bigint}, \texttt{\textbackslash real}\} \\ \min \tilde{Z} & \text{otherwise} \end{cases}$$

- $val_s(\texttt{\textbackslash num\_of nullable}\ T\ x; a; b) = val_s(\texttt{\textbackslash sum nullable}\ T\ x; a\ \texttt{\&\&}\ b; \texttt{1L})$

Where $\bigoplus$ and $\bigotimes$ denote addition and multiplication according to the semantics of type T' (i.e. including possible overflows). `1L` denotes the constant 1 with dynamic type long, thus enforcing respective overflow semantics.

- $wd(\mathcal{Q}$ `nullable` $T\ x; b; e)$ **iff** $wd(b)$ for every value of x, $wd(e)$ and $|Z| < \infty$ for $\mathcal{Q} \in \{$`\sum`, `\product`, `\num_of`$\}$
- $wd(\mathcal{Q}$ `nullable` $T\ x; b; e)$ **iff** $wd(b)$ for every value of x, $wd(e)$, $|Z| < \infty$ and if $T' \in \{$`\bigint`, `\real`$\}$ then $Z \neq \emptyset$ for $\mathcal{Q} \in \{$`\max`, `\min`$\}$
- $\omega(\mathcal{Q}$ `nullable` $T\ x; b; e) = id$ for every generalized quantifier $\mathcal{Q}$

Appendix B

What's Missing?

B.1 Omitted Java features

Java 5

One assumption for this work is, that the specified programming language is Java as of version 1.4. In particular, only non-generic types are considered. As of the time of writing, the Java Modeling Language has yet no support for generics. An adaptation of JML to Java 6 has been recently proposed [Cok08].

Static initialization

Static initialization of classes [GJSB00, Sect. 8.7] is not considered in this work for reasons of simplicity. In fact, it is never easy to tell when exactly classes are initialized, especially if initialization occurs during the evaluation of expressions. Consider for instance an access on a static field in the precondition of a method specification. At first, we need to apply case distinction whether the class is yet initialized and if, invoke the initializer and evaluate the location in the post-state of the initializer. This again raises the question whether the class is initialized in the *post-state* of the method in question, since it was not initialized in the pre-state (but only during evaluation of the specification).

Static initialization has effects on semantics, however small: States are only visible for a type if it has finished static initialization. In our approach, this is always given since we require all classes to be fully initialized. The predicate `\is_initialized` has been omitted from the expression reference since it would trivially yield true in every state.

Reachability and finalizers

Our approach only covers a part of the live-cycle of objects in Java. A basic notion of reachability could be included, it is though not necessary. Reachability itself has to effect on the semantics of JML since all specifications are defined for any object – regardless of reachability. On the "Java hand", an unreachable object will not only be removed from memory by garbage collection eventually (which also does not matter to us); but it will also invoke *finalizers* [GJSB00, Sect. 12.6]. Finalizers are problematical for three reasons:

- Finalizers run concurrently to the main thread and each other. We decided not to cover concurrency (see below).
- Though finalizers are only invoked on unreachable objects, they may refer to reachable ones. This includes making the receiver reachable again as well as making other objects unreachable. As it can be seen in [GJSB00, p. 246], the complete life-cycle model includes 11 states.
- It is not clear, *exactly when* a finalizer is invoked after an object has become unreachable. This can be seen as a kind of indeterministic program flow.

B.2 JML features not covered

Real-time and concurrency issues

Neither real-time nor concurrency specifications appear in this work. This would perhaps fill a thesis of its own. Even the reference manual admits that the current version of JML focusses on sequential programs. Not covered are thus the `monitors_for`, `duration`, `working_space` and `when` clauses in type, resp. method specifications as well as the following expressions: `\duration`, `\space`, `\working_space`, `\max`, `\lockset` and the lockset order predicate.

Model methods and model types

In JML, not only fields may be added for means of specification, but also methods and types. The interpretation of model methods [SLN07] and types [Cha06, DM07b] is an issue of on-going research. There is no approach to describe model types in general, yet. In [DM07b], every model type is statically mapped to an entity in another formal language (in this case Isabelle). E.g. `JMLObjectSet` is mapped to Isabelle's set representation. Even the JML run-time checker [JML08] is yet incapable of coping with model types.

JML's rendition of (finite) sets, sequences, functions and other mathematical entities are based on model types and model methods and are therefore left out.

The Universe Type System

The *Universe* Type System [MP01, Mül02] is an ownership type system. It defines *owners* for references to objects through which they may be accessed. This might be used, for instance, to create a more modular definition of invariants. It heavily affects the type system, in that for every type T of our definition in Sect. 2.2 there would be three types instead: `rep` T, `peer` T, and `readonly` T.

Math modes

As already explained in Sect. 3.3.2, there are three different arithmetical modes which define different semantics of numerical expressions. JML allows them to be switched from one class to another, e.g. the invariants of a class C modified with `spec_save_math` are defined with the safe math extension, while a method of class D, which uses the default Java arithmetic, has to respect that invariant. This seems needlessly complicated, so we do not concern switching math modes.

`readable` and `writable` clauses

The `readable` and `writable` clauses are type specifications which name necessary conditions for accessing resp. assigning fields. The main difference to `accessible`/`assignable` clauses of method specifications is, that these conditions have to hold in the very state in which the field is accessed. Furthermore, this definition is very different to the other type specifications in that is has to hold in every state, not just visible ones.

Therefore, `readable` and `writable` cannot be desugared to any other specification element (method specifications, invariants or history constraints). Although it would not be very complicated to add another specification element to Sect. 4.1, we refrain from this idea because it would not reveal many new insights.

The keyword `for` (history constraints)

A history constraint may be weakened in that it only has to be preserved by the (non-helper) methods named after the `for` argument. The default is `\everything`, which is assumed in Sect. 4.1.3.

`forall` and `old` clauses

In method specifications, variables may be bound by `forall` or `old` clauses which scope is the whole specification case (i.e. all clauses within the current `behavior`). The intuitive meaning of `forall` is that the specification is valid for every valuation of the named variables. Since specifications are already meant to be valid for any pre-state, this is purely a syntactical feature.

`old` $T\ v = x$ binds a variable v to the pre-state value of an expression x for later use in one of the postconditions. This is again purely syntactical since we can substitute `\old`(x) for v in the postcondition.

`callable` clause

Similar to an `assignable` clause, method specifications can bear a `callable` clause which lists all methods which may be (directly or indirectly) invoked. To extend our model to include an interpretation for this clause, we would have to resolve the listed method identifiers and check whether they occur on any call stack of the run.

`measured_by` clause

The `measured_by` clause is used for the proof of termination for recursive or mutually recursive methods. It takes an expression of type `int` which must decrease at every method invocation during a run. To extend our model to include an interpretation for this clause, we would have to investigate any call stack of the run and assert the decrease in every state in which a method has just been pushed on the stack. (See [HLL$^+$09, Sect. 2.3.5].)

`captures` clause

In a method specification, it can be specified whether objects referenced by parameters are *captured* by the method. An object is captured if its reference appears on the right-hand side of an assignment. As assignments are within the domain of the black box in our approach, there is no way of describing this clause formally.

Refining statements

The keyword `refining` introduces the possibility of annotating any (possibly annotated) statement block. The specification grammar is identical to a method specification. In this way, there may be given frame conditions to loops (or any other program fragment), for instance.

`\old` with labels

Within assertions in annotations, the `\old` may be used not only to refer to the pre-state, but also to intermediate states which are reached in the execution of program code carrying a label. By this, expressions would not be evaluated with respect to two states (pre-state and current), but to an arbitrary number.

The `\invariant_for` expression

The expression `\invariant_for`(x) for an expression x of reference type is meant to tell whether $val(x)$ satisfies the invariant of the static type of x. Of course, this only makes sense for states which are not visible for $val(x)$. The problem with this expression is, that is not possible to statically deduce the exact run-time type of an expression. E.g. the expression `\invariant_for((Number) new Integer())` is meant to refer to the invariant of type `Number`, but the expression `(Number) new Integer()` is semantically equivalent to `new Integer()`.

The `\reach` expression

The `\reach` expression is used to tell which objects are currently reachable from some reference. In theory, is a very sensible expression. It however relies on the model type `JMLObjectSet` which we excluded from our considerations (see above).

```
    public class Node {
        private int value;
        private /*@ nullable @*/ Node leftChild;
        private /*@ nullable @*/ Node rightChild;

        public static Node root;

        /*@ private invariant (\forall Node n;
          @              \reach(root); n.value > 0);
          @*/
    }
```

Figure B.1: A possible alternative definition of object reachability without using model types.

In our opinion, it would generally be a good idea to give another definition of reachability which does not depend on model types. One could think of a predicate which can be used as range of quantification. E.g. the specification in Fig. B.1 for a binary tree is meant to span over the nodes of the tree.

Informal predicates, redundancy, debugging etc.

There is a variety of JML features which do not have an effect on formal semantics, but are used to describe specifications in an informal and intuitive way. This is mostly to give the viewer, e.g. a programmer, some intuition on what the formal specifications means, e.g. through an example.

To this category belong "informal predicates", which are just common comments within the program code. With the keywords `hence_by`, `for_example` and *A*`_redundantly` (where *A* is a specification clause such as `requires`) logical conclusions or examples can written to an existing formal specification. E.g. one could conclude that the factorial of a non-negative number is strictly greater than zero:

```
/*@ public normal_behavior
  @     requires x >= 0;
  @     ensures \result ==
  @         (\product int z; 0 < z & z <= x; z);
  @     ensures_redundantly \result > 0;
  @*/
public int factorial (int x);
```

Furthermore, there are statements which are used for debugging and labeling with a run-time checker. They possess, of course, no formal semantics.

Bibliography

[ABHS07] Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. KeY: A formal method for object-oriented systems. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4468 of *Lecture Notes in Computer Science*, pages 32–43. Springer, 2007.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, 1996.

[ASM80] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. Specification language. In *On the Construction of Programs*, pages 343–410. Cambridge University Press, 1980.

[BBS01] Thomas Baar, Bernhard Beckert, and Peter H. Schmitt. An extension of dynamic logic for modelling OCL's *@pre* operator. *Lecture Notes in Computer Science*, 2244:47–54, 2001.

[BCC$^+$05] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

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

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[Blo99] Joshua Bloch. A simple assertion facility. Java Specification Request 41, Java Community Process, November 1999.

[Blo01] Joshua Bloch. *Effective Java: Programming Language Guide*. The Java Series. Addison-Wesley, 2001.

[BNSS05] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specification. In *ECOOP Workshop FTfJP'2004 Formal Techniques for Java-like Programs*, pages 51–60, January 2005.

[BP03] Cees-Bart Breunesse and Erik Poll. Verifying JML specifications with model fields. In *Formal Techniques for Java-like Programs (FTfJP)*, number 408 in Technical Report, ETH Zurich, pages 51–60, July 2003.

[BvW98] Ralph Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, Germany, 1998.

[Cha03] Patrice Chalin. Improving JML: For a safer and more effective language. Technical Report 2003-001.1, Computer Science Department, Concordia University, March 2003.

[Cha04] Patrice Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. *Journal of Object Technology*, 3(6):57–79, June 2004. Special issue: ECOOP 2003 Workshop on FTfJP.

[Cha06] Julien Charles. Adding native specifications to JML. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, July 2006.

[Cha07] Patrice Chalin. A sound assertion semantics for the dependable systems evolution verifying compiler. In *ICSE*, pages 23–33. IEEE Computer Society, 2007.

[CKLP06] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and*

Objects (FMCO) 2005, Revised Lectures, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer-Verlag, 2006.

[CL94] Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, July 1994.

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

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

[Cok08] David Cok. Adapting JML to generic types and Java 1.6. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 27–34, 2008.

[CR06] Patrice Chalin and Frédéric Rioux. Non-null references by default in the Java Modeling Language. *ACM SIGSOFT Software Engineering Notes*, 31(2), 2006.

[DM06] Á. Darvas and P. Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, June 2006.

[DM07a] Á. Darvas and P. Müller. Formal encoding of JML Level 0 specifications in JIVE. Technical Report 559, ETH Zurich, 2007. Annual Report of the Chair of Software Engineering. 17 pages.

[DM07b] Ádám Darvas and Peter Müller. Faithful mapping of model classes to mathematical structures. In *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 31–38. ACM, September 2007.

[Eng05] Christian Engel. A translation from JML to JavaDL. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe, February 2005.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[ER07] Christian Engel and Andreas Roth. KeY quicktour for JML. `http://www.key-project.org/download/quicktour/quicktourJML-1.2.pdf`, 2007.

[GHW85] J. Guttag, J. Horning, and J. Wing. The larch family of specification languages. *IEEE Software*, 2(5):24–36, 1985.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition.* Addison Wesley, 2000.

[GM95] James Gosling and Henry McGilton. *The Java Language Environment: a white paper.* Sun Microsystems, October 1995.

[GS95] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. *Lecture Notes in Computer Science*, 1000:366–373, 1995.

[Gut97] Scott B. Guthery. Java Card: Internet computing on a smart card. *IEEE Internet Computing*, 1(1):57–59, 1997.

[Häh05] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005.

[Har84] David Harel. Dynamic logic. In Dov Gabbay and F. Guenthner, editors, *Handbook of philosophical logic*, chapter II.10, pages 497–604. Reidel, 1984.

[HLL+09] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, School of EECS, Orlando, FL, March 2009.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[JML08] Common JML tools. Downloadable from `http://www.eecs.ucf.edu/~leavens/JML/download.shtml`, December 2008. Version 5.6_rc3.

[JP01] Bart Jacobs and Erik Poll. A logic for the java modeling language JML, January 24 2001.

[Kah87] W. Kahan. Doubled-precision IEEE standard 754 floating-point arithmetic. Manuscript, February 1987.

[KTB91] B. Konikowska, A. Tarlecki, and A. Blikle. A three–valued logic for software specification and validation. *Fundamenta Informaticae*, 14:411–453, 1991.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[LBR03] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science, 2003. Revised May 2006.

[LC05] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from `http://www.jmlspecs.org/`, 2005.

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

[Lea99] Gary T. Leavens. Larch/C++ reference manual. Available from `ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz`, April 1999.

[Lea06] Gary T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Zhiming Liu and He Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, November 2006. Springer-Verlag.

[Lea09] Gary T. Leavens, February 2009. Posting on mailing list `jmlspecs-interest@lists.sourceforge.net`.

[Leh09] Hermann Lehner, March 2009. Posting on mailing list `jmlspecs-interest@lists.sourceforge.net`.

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

[LM06] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130, New York, NY, March 2006. Springer-Verlag.

[LPC$^+$08] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual. Draft version, revision 1.231, Available from `http://www.jmlspecs.org/`, May 2008.

[Łuk20] Jan Łukasiewicz. O logice tròjwartośćiowej. *Ruch Filozoficzny*, 5:169–171, 1920.

[LW93] Barbara Liskov and Jeanette M. Wing. Specifications and their use in defining subtypes. In Andreas Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 16–28, Washington DC, USA, 1993. ACM Press.

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

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison Wesley Longman, Inc., second edition, April 1999.

[Mey92a] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[Mey92b] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

[MMP97] Peter Müller, Jörg Meyer, and Arnd Poetzsch-Heffter. Programming and interface specification language of JIVE - specification and design rationale. Technical report, Fernuniversität Hagen, December 05 1997.

[MP01] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[Mül02] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2002.

[OCL05] Object Modeling Group. *Object Constraint Language Specification, version 2.0*, June 2005.

[ORR$^+$96] Sam Owre, Sreeranga Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In *International Conference on Computer Aided Verification (CAV) , New Brunswick, New Jersey*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, July 1996.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[PM98] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, pages 404–423, 1998.

[Poe97] Arndt Poetzsch-Heffter. *Specification and verification of object-oriented programs.* Habilitationsschrift, Technische Universität München, January 1997.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[RL00] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical report, Iowa State University, August 08 2000.

[SLN07] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. Technical Report 07-04b, Iowa State University, Department of Computer Science, July 2007.

[SR05] A. Sălcianu and M. Rinard. Purity and side effect analysis for Java programs. *Lecture Notes in Computer Science*, 3385:199–215, 2005.

[SSB01] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: definition, verification, validation.* Springer-Verlag, 2001.

[Sun04] Sun Microsystems. *Java 2 Platform, Standard Edition 5.0: API Specification*, 2004.

[Ulb08] Mattias Ulbrich. A set-theoretic model for java states. unpublished, 2008.

[Wei07] Benjamin Weiß. Inferring invariants by static analysis in KeY. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2007.

List of Symbols

$\mathcal{C}$	set of classes and interfaces, 11
$\mathcal{D}$	data group, 46
$\mathcal{E}$	set of expressions, 28
$\mathcal{J}$	black box function, 22
$\mathcal{L}$	set of (concrete) locations, 15
$\mathcal{L}'$	set of locations, incl. abstract, 40
$\mathcal{N}$	normalization (super-script), 29
$\mathcal{P}$	set of packages, 12
$\mathcal{S}$	set of system states, 18
$\mathcal{T}$	set of types, 13
$\mathcal{U}$	universe, 15
$\mathcal{V}_R$	visibility relation, 69
α	annotation function, 22
α^*	updated annotation function, 85
χ	call stack, 18
λ_a	accessed locations, 22
λ_w	assigned *(written)* locations, 22
Ω	termination mode, 22
ω	successor function, 34
Π	program, 11
π^A	annotated code fragment, 12
Σ	logical state, 35
σ	Stack, 16
h	Heap, 16
loc	location valuation function, 45
M	model field valuator, 39
msp	most-specific method, 18
R^*	updated run, 85
s^*	updated state, 85
V_T	domain (compatible elements), 13
V_T^-	domain (compatible elements, excl. null), 15

V_T^0	domain (direct instances), 14
val	valuation function, 28
wd	well-definition predicate, 28
$\mathbb{I}$	set of identifiers, 12
$\mathfrak{M}$	set of model field characterizations, 37
$\sqsubseteq$	subtype relation, 13
$\models$	validity relation (without axioms), 35
$\Vvdash$	validity relation (with axioms), 43
`<:`	subtype relation (JML expression), 57

Index