Appendix A
Java Modeling Language Reference

Daniel Grahl

This appendix serves as a comprehensive reference for the syntax and semantics the dialect of the Java Modeling Language (JML) that is used in the KeY system, version 2.6. The extensions for information flow introduced in Section 13.4 are also included. Section A.1 presents the full syntax of JML as it is supported by the KeY system. In Section A.2, the semantics of JML expressions is given through a translation to JavaDL. Refer to Chapters 7ff. for in depth explanations and a discussion on these items. Contract semantics are not covered here; they are treated extensively in Section 8.2. We cover the issue of well-definedness in Section A.3.

A.1 JML Syntax

The syntax of JML is heavily intertwined with the syntax of the Java language. The JML reference manual [Leavens et al., 2013] presents a complete grammar for Java enriched with (standard) JML specifications. This includes every grammatical feature—from lexical tokens to compilation units. Here, in this appendix, we take on another approach in not reiterating the entire Java syntax, but indicating where the given definitions would be injected into it. This section only describes syntax; we assume that it is clear from the context how to produce expressions that are well-typed; in doubt, refer to the JML reference manual [Leavens et al., 2013].

Since JML has been designed for other analysis approaches, not all constructs that are introduced in the JML reference manual [Leavens et al., 2013]do make sense for the KeY approach. We present only that part of the syntax for which we can give a formal semantics. Please note that the targeted subset of Java does not include all Java 5–8 features, such as autoboxing, generics, etc. We also omit the JML delimiters `//@` (end of line style) and `/*@ @*/` (block style). We structure this section after the different locations where JML specifications may appear.

The grammar is given in a Backus Naur style using the postfix operators $^?$ (optional occurrence), * (any number of occurrences), and $^+$ (at least one occurrence).

Table A.1 Additional class members in JML

JML Syntax

⟨*ClassElem*⟩ ::= ⟨*ClassSpec*⟩ | ⟨*MMthd*⟩
⟨*ClassSpec*⟩ ::= ⟨*Visibility*⟩$^?$ (⟨*ClassInv*⟩ | ⟨*FieldDecl*⟩ | ⟨*Represents*⟩ | ⟨*MAccess*⟩) ;
⟨*ClassInv*⟩ ::= ⟨*StaticOrInstance*⟩$^?$ ⟨*ClassInvKw*⟩ ⟨*BoolExpr*⟩
⟨*StaticOrInstance*⟩ ::= `static` | `instance`
⟨*ClassInvKw*⟩ ::= `invariant` | `constraint` | `initially` | `axiom`
⟨*FieldDecl*⟩ ::= (`ghost` | `model`) ⟨*Type*⟩ ⟨*Id*⟩$^+$
⟨*Represents*⟩ ::= `represents` ⟨*Id*⟩ `=` ⟨*Expr*⟩ ; | `represents` ⟨*Id*⟩ `\such_that` ⟨*BoolExpr*⟩ ;
⟨*MAccess*⟩ ::= `accessible` ⟨*Id*⟩ `:` ⟨*LocSetExpr*⟩ ⟨*Mby*⟩$^?$;
⟨*MMthd*⟩ ::= ⟨*Contract*⟩ (`no_state` | `two_state`)$^?$ `model` ⟨*Type*⟩ ⟨*Id*⟩ `(` ⟨*Params*⟩ `)`
`{ return` ⟨*Expr*⟩`; }`
⟨*Params*⟩ ::= (⟨*Type*⟩ ⟨*Id*⟩ (`,` ⟨*Type*⟩ ⟨*Id*⟩)*)$^?$

JML Syntax

In Table A.1, the rule ⟨*ClassElem*⟩ refers to elements which may additionally appear as a Java class member (see [Gosling et al., 2013, Sect. 8.2]). Rule ⟨*Id*⟩ refers to valid identifiers in Java (see ibid., Sect. 6) for types (i.e., classes or interfaces), variables, fields, and methods.

Table A.2 Contract grammar in JML

JML Syntax

⟨*Contract*⟩ ::= `also`$^?$ ⟨*SpecCase*⟩ (`also` ⟨*SpecCase*⟩)*
⟨*SpecCase*⟩ ::= ⟨*Visibility*⟩$^?$ ⟨*Behavior*⟩$^?$ ⟨*Clause*⟩*
⟨*Visibility*⟩ ::= `public` | `protected` | `private`
⟨*Behavior*⟩ ::= `normal_behavior` | `normal_behaviour` |
`exceptional_behavior` | `exceptional_behaviour`
⟨*Clause*⟩ ::= (⟨*Requires*⟩ | ⟨*Ensures*⟩ | ⟨*Signals*⟩ | ⟨*SignalsOnly*⟩ | ⟨*Diverges*⟩ |
⟨*Determs*⟩ | ⟨*Assign*⟩ | ⟨*Acc*⟩ | ⟨*Mby*⟩) ; | `{|` ⟨*Clause*⟩* `|}`
⟨*Requires*⟩ ::= (`requires` | `pre`) ⟨*BoolExpr*⟩ ;
⟨*Ensures*⟩ ::= (`ensures` | `post`) ⟨*BoolExpr*⟩ ;
⟨*Signals*⟩ ::= (`signals` | `ensures`) (⟨*Type*⟩ ⟨*Id*⟩$^?$) ⟨*BoolExpr*⟩ ;
⟨*SignalsOnly*⟩ ::= ⟨*Type*⟩ (`,` ⟨*Type*⟩)* | `\nothing` | `\everything` ;
⟨*Diverges*⟩ ::= `diverges` ⟨*BoolExpr*⟩ ;
⟨*Determs*⟩ ::= `determines` (⟨*Exprs*⟩ | `\nothing`) `\by` (⟨*Exprs*⟩ | `\itself` | `\nothing`)
(`\declassifies` ⟨*Exprs*⟩ | `erases` ⟨*Exprs*⟩)$^?$ (`\new_objects` ⟨*Exprs*⟩)$^?$;
⟨*Assign*⟩ ::= ⟨*AssignKw*⟩ (⟨*LocSetExpr*⟩ (`,` ⟨*LocSetExpr*⟩)* |
`\nothing` | `\strictly_nothing` | `\everything`) ;
⟨*AssignKw*⟩ ::= `assignable` | `modifiable` | `modifies`
⟨*Acc*⟩ ::= `accessible` (⟨*LocSetExpr*⟩ (`,` ⟨*LocSetExpr*⟩)* | `\nothing` | `\everything`) ;
⟨*Mby*⟩ ::= `measured_by` ⟨*Exprs*⟩ ;

JML Syntax

Table A.2 shows the grammar used for method contracts. Contracts can appear immediately before method declarations (modulo whitespace; see ibid., Sect. 8.4).

The standard JML clauses interpreted by KeY are limited to **requires**, **ensures**, **signals**, **signals_only**, **diverges**, **assignable**, **accessible**, **measured_by**. For all of these, their synonyms are defined, too (not all are shown here). Other clauses can still be parsed in KeY, but will be ignored. Additionally, the **determines** clause (see Section 13.4) extends standard JML.

Table A.3 shows modifiers in addition to Java's modifiers (see ibid., Sects. 8.1.1, 8.3.1, 8.4.3, 9.1.1). Note that, technically, the **nullable** and **non_null** are also modifiers, but for more readable semantics we treat them as type information.

Table A.3 Modifiers in JML

JML Syntax

⟨*Mod*⟩ ::= **pure** | **strictly_pure** | **helper** | **model** | **nullable_by_default**

JML Syntax

Table A.4 shows annotations that may appear inside method bodies, such as loop invariants, block contracts [Wacker, 2012], or ghost assignment statements. Set statements may appear whenever the Java language expects a statement [Gosling et al., 2013, Sect. 14.5], block contracts may appear immediately before a statement block, and loop invariants must appear immediately before a loop statement (**while**, (enhanced) **for**, or **do** loops; see ibid., Sects. 14.12ff.). As a technical restriction of the implementation of KeY, any annotation statement must not be the last statement in a statement block.

Table A.4 JML annotation grammar

JML Syntax

⟨*Annot*⟩ ::= ⟨*LoopInv*⟩ | ⟨*BlockCntr*⟩ | ⟨*SetStm*⟩ | ⟨*Assert*⟩ | **unreachable;**
⟨*LoopInv*⟩ ::= (⟨*LoopInvClause*⟩**;**)$^+$ (⟨*VariantClause*⟩**;**)$^?$ (⟨*Assign*⟩**;**)$^?$ (⟨*Determs*⟩**;**)$^?$
⟨*LoopInvClause*⟩ ::= (**maintaining** | **loop_invariant**) ⟨*BoolExpr*⟩
⟨*VariantClause*⟩ ::= (**decreasing** | **decreases**) ⟨*Exprs*⟩
⟨*BlockCntr*⟩ ::= **also**$^?$ ⟨*BSpecCase*⟩ (**also** ⟨*BSpecCase*⟩)*
⟨*BSpecCase*⟩ ::= ⟨*BBehavior*⟩$^?$ ⟨*BClause*⟩*
⟨*BBehavior*⟩ ::= **behavior** | **normal_behavior** | **exceptional_behavior** |
break_behavior | **continue_behavior** | **return_behavior**
⟨*BClause*⟩ ::= ⟨*Breaks*⟩**;** | ⟨*Returns*⟩**;**| ⟨*Clause*⟩
⟨*Breaks*⟩ ::= (**breaks** | **continues**) ⟨*Id*⟩$^?$ ⟨*BoolExpr*⟩
⟨*Returns*⟩ ::= **returns** ⟨*BoolExpr*⟩
⟨*SetStm*⟩ ::= **set** ⟨*Loc*⟩ **=** ⟨*Expr*⟩**;**
⟨*Assert*⟩ ::= **assert** ⟨*BoolExpr*⟩**;**

JML Syntax

Table A.5 shows the grammar for expressions in JML. This treatment is complete; all other Java expressions are not valid in JML and thus are rejected by KeY's parser. Rule ⟨*Literal*⟩ refers to (integer) literals of any type or numeral system (see ibid., Sect. 15.8.1).

Table A.5 JML expression grammar

JML Syntax

⟨*Exprs*⟩ ::= (⟨*Expr*⟩`,`)* ⟨*Expr*⟩
⟨*Expr*⟩ ::= ⟨*BoolExpr*⟩ | ⟨*IntExpr*⟩ | ⟨*LocSetExpr*⟩ | ⟨*SeqExpr*⟩ | `this` | `null`
⟨*BoolExpr*⟩ ::= `!`⟨*BoolExpr*⟩ | ⟨*BoolExpr*⟩⟨*BinaryBoolOp*⟩⟨*BoolExpr*⟩ |
⟨*Expr*⟩⟨*EqOp*⟩⟨*Expr*⟩ | ⟨*IntExpr*⟩⟨*CompareOp*⟩⟨*IntExpr*⟩ |
`(`⟨*Quant*⟩ ⟨*Type*⟩ ⟨*Id*⟩+`;` (⟨*BoolExpr*⟩`;`)? ⟨*BoolExpr*⟩`)` |
`\invariant_for(`⟨*Expr*⟩`)` | `\static_invariant_for(`⟨*Type*⟩`)` |
`\fresh(`⟨*Expr*⟩`)` | `\nonnullelements(`⟨*Expr*⟩`)` |
`\is_initialized(\type(`⟨*Type*⟩`))` |
`\reach(`⟨*Id*⟩`,` ⟨*Expr*⟩`,` ⟨*Expr*⟩(`,` ⟨*IntExpr*⟩)?`)` | ⟨*Expr*⟩ `instanceof` ⟨*Type*⟩ |
`\typeof(`⟨*Expr*⟩`)` ⟨*TypeOp*⟩ `\type(`⟨*Type*⟩`)` | `\subset(`⟨*SeqExpr*⟩`,` ⟨*SeqExpr*⟩`)` |
`\disjoint(`⟨*SeqExpr*⟩ (`,` ⟨*SeqExpr*⟩)+`)`| `\new_elems_fresh(`⟨*LocSetExpr*⟩`)` |
`true` | `false` | ⟨*GenExpr*⟩
⟨*BinaryBoolOp*⟩ ::= `&&` | `&` | `||` | `|` | `==>` | `<==` | `<==>` | `<=!=>` | `^`
⟨*EqOp*⟩ ::= `==` | `!=`
⟨*CompareOp*⟩ ::= `<` | `<=` | `>` | `>=`
⟨*TypeOp*⟩ ::= `==` | `<:`
⟨*Quant*⟩ ::= `\forall` | `\exists`
⟨*IntExpr*⟩ ::= `-`⟨*IntExpr*⟩ | `~`⟨*IntExpr*⟩ | ⟨*IntExpr*⟩⟨*BinaryIntOp*⟩⟨*IntExpr*⟩ |
⟨*IntExpr*⟩⟨*BinaryIntOpBitw*⟩⟨*IntExpr*⟩ | ⟨*Comprehension*⟩ | `\index` |
⟨*Expr*⟩`.length` | ⟨*Literal*⟩| ⟨*GenExpr*⟩
⟨*BinaryIntOp*⟩ ::= `+` | `-` | `*` | `/` | `%`
⟨*BinaryIntOpBitw*⟩ ::= `<<` | `>>` | `>>>` | `&` | `|` | `^`
⟨*Comprehension*⟩ ::= `(`⟨*ComprOp*⟩ ⟨*Type*⟩ ⟨*Id*⟩+`;` (⟨*BoolExpr*⟩`;`)? ⟨*IntExpr*⟩`)` |
`(\num_of` ⟨*Type*⟩ ⟨*Id*⟩+`;` (⟨*BoolExpr*⟩`;`)? ⟨*IntExpr*⟩`)`
⟨*ComprOp*⟩ ::= `\sum` | `\product` | `\max` | `\min`
⟨*GenExpr*⟩ ::= ⟨*BoolExpr*⟩`?` ⟨*Expr*⟩`:` ⟨*Expr*⟩ | `\result` | `\old(`⟨*Expr*⟩`)` |
`\pre(`⟨*Expr*⟩`)` | `(\lblneg` ⟨*Id*⟩ ⟨*Expr*⟩`)` | `(\lblpos` ⟨*Id*⟩ ⟨*Expr*⟩`)` |
`\dl_`⟨*Id*⟩`(`⟨*Exprs*⟩?`)` | `\exception` |
`(`⟨*Type*⟩`)` ⟨*Expr*⟩ | ⟨*Loc*⟩ | (⟨*Expr*⟩`.`)?⟨*Id*⟩`(`⟨*Exprs*⟩?`)` |
`(`⟨*Type*⟩`)` ⟨*SeqExpr*⟩`[`⟨*IntExpr*⟩`]` | `(*` ⟨*JavaDLTerm*⟩ `*)`

⟨*Loc*⟩ ::= (⟨*Expr*⟩`.`)?⟨*Id*⟩ | ⟨*Type*⟩`.`⟨*Id*⟩ | ⟨*Expr*⟩`[`⟨*IntExpr*⟩`]`
⟨*LocSetExpr*⟩ ::= ⟨*Expr*⟩`.`⟨*Id*⟩ | ⟨*Expr*⟩`[`⟨*IntExpr*⟩`]` | ⟨*Expr*⟩`[`⟨*IntExpr*⟩`..`⟨*IntExpr*⟩`]`|
⟨*Expr*⟩`[*]` | ⟨*Expr*⟩`.*` | `\empty` | `\everything` |
⟨*LocSetOp*⟩`(`⟨*LocSetExpr*⟩`,` ⟨*LocSetExpr*⟩`)` |
`\infinite_union(`⟨*Type*⟩ ⟨*Id*⟩`;` (⟨*BoolExpr*⟩`;`)?⟨*LocSetExpr*⟩`)` |
`\reachLocs(`⟨*Id*⟩`,` ⟨*Expr*⟩(`,` ⟨*IntExpr*⟩)?`)` | ⟨*GenExpr*⟩
⟨*LocSetOp*⟩ ::= `\intersect` | `\set_union` | `\set_minus`
⟨*SeqExpr*⟩ ::= `\seq_empty` | `\seq_singleton(`⟨*Expr*⟩`)` | `\values` |
`\seq_concat(`⟨*SeqExpr*⟩`,` ⟨*SeqExpr*⟩`)` |
⟨*SeqExpr*⟩`[`⟨*IntExpr*⟩`..`⟨*IntExpr*⟩`]` | ⟨*GenExpr*⟩
`(\seq_def` ⟨*Type*⟩ ⟨*Id*⟩`;`⟨*IntExpr*⟩`;` ⟨*IntExpr*⟩`;` ⟨*Expr*⟩`)`
⟨*Type*⟩ ::= `boolean` | `byte` | `char` | `short` | `int` | `long` | `\bigint` | `\seq` |
`\locset` | ⟨*NullMod*⟩? ⟨*Id*⟩(`[]`)*
⟨*NullMod*⟩ ::= `nullable` | `non_null`

JML Syntax

A.2 JML Expression Semantics

This section contains semantics for JML expressions, given as translations into JavaDL. Tables A.6–A.8 contain translations for Boolean expressions (yielding formulas). For some of the logical operators two or more alternatives with the same semantics exist.[1] All other tables contain translations to terms. See Section 8.1 for details.

Table A.6 Translation of Boolean JML operators

$e \in$ JExp (alternatives)		**JavaDL** $\lfloor e \rfloor \in$ DLFml
`!`A		$\neg\lfloor A\rfloor$
A `&&` B	A `&` B	$\lfloor A\rfloor \wedge \lfloor B\rfloor$
A `||` B	A `|` B	$\lfloor A\rfloor \vee \lfloor B\rfloor$
A `==>` B	B `<==` A	$\lfloor A\rfloor \rightarrow \lfloor B\rfloor$
A `<==>` B	A `==` B	$\lfloor A\rfloor \leftrightarrow \lfloor B\rfloor$
A `<=!=>` B	A`^`B A `!=` B	$\neg(\lfloor A\rfloor \leftrightarrow \lfloor B\rfloor)$
`(\forall` T `x1, ... xn;` A`;` B`)`		$\forall \lfloor T\rfloor x_1; \ldots \forall \lfloor T\rfloor x_n;\ (\bigwedge_{i=1}^{n} \text{inRange}_T(x_i) \wedge \lfloor A\rfloor \rightarrow \lfloor B\rfloor)$
`(\exists` T `x1, ... xn;` A`;` B`)`		$\exists \lfloor T\rfloor x_1; \ldots \exists \lfloor T\rfloor x_n;\ (\bigwedge_{i=1}^{n} \text{inRange}_T(x_i) \wedge \lfloor A\rfloor \wedge \lfloor B\rfloor)$

Table A.7 Translation of special Boolean JML operators

JML expression e	**JavaDL** $\lfloor e \rfloor \in$ DLFml
`\invariant_for(`o`)`	$Object$::inv(`heap`, $\lfloor o\rfloor$)
`\static_invariant_for(`T`)`	$\lfloor T\rfloor$::\$$inv$(`heap`)
`\fresh(`o`)`	$select_{boolean}$(`heap`^{pre}, $\lfloor o\rfloor$, $created$) $\doteq$ $FALSE \wedge \lfloor o\rfloor \neq$ `null`
`\nonnullelements(`a`)`	$\lfloor a\rfloor \neq$ `null` $\wedge \forall i.(0 \leq i < \lfloor a\rfloor.length \rightarrow select_{Object}(\text{heap}, \lfloor a\rfloor, arr(i)) \neq$ `null`$)$
`\is_initialized(\type(`T`))`	$\lfloor T\rfloor$`.<classInitialised>` $\doteq$ $TRUE$
`\reach(f, o1, o2, n)`	$reach$(`heap`, $allObjects$(`f`), $\lfloor o_1\rfloor$, $\lfloor o_2\rfloor$, $\lfloor n\rfloor$)
`\reach(f, o1, o2)`	$\exists$ int n; $reach$(`heap`, $allObjects$(`f`), $\lfloor o_1\rfloor$, $\lfloor o_2\rfloor$, n)
`\typeof(`x`) == \type(`T`)`	$exactInstance_{\lfloor T\rfloor}(\lfloor x\rfloor)$
x `instanceof` T	$instance_{\lfloor T\rfloor}(\lfloor x\rfloor)$

Table A.8 Predicates on location sets

JML expression e	**JavaDL** $\lfloor e \rfloor \in$ DLFml
`\subset(`s`,`t`)`	$subset(\lfloor s\rfloor, \lfloor t\rfloor)$
`\disjoint(`s_1`, ..., `s_n`)`	$\bigwedge_{1\leq i<j\leq n} disjoint(\lfloor s_i\rfloor, \lfloor s_j\rfloor)$
`\fresh(`s`)`	$subset(\lfloor s\rfloor, unusedLocs(\text{heap}^{pre}))$
`\new_elems_fresh(`s`)`	$subset(\lfloor s\rfloor, union(\{\text{heap} := \text{heap}^{pre}\}\lfloor s\rfloor, unusedLocs(\text{heap}^{pre})))$

Table A.9 Translation of Java integer expressions. Depending on the type of subexpressions, there are different translations: The right-most column applies if at least one operand is of type `\bigint`, else if at least one operand is of type `long`, the center column applies, and else the left column applies. Note that there is no cast to `\bigint`. Refer to Section 5.4 for a detailed explanation on these functions.

$e \in \text{JExp}$	$\lfloor e \rfloor \in \text{Trm}_{int}$ $type_{\text{JML}}(e) = \texttt{int}$	$\lfloor e \rfloor \in \text{Trm}_{int}$ $type_{\text{JML}}(e) = \texttt{long}$	$\lfloor e \rfloor \in \text{Trm}_{int}$ $type_{\text{JML}}(e) = \texttt{\bigint}$
`- n`	$javaUnaryMinusInt(\lfloor n \rfloor)$	$javaUnaryMinusLong(\lfloor n \rfloor)$	$-\lfloor n \rfloor$
`n + m`	$javaAddInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaAddLong(\lfloor n \rfloor, \lfloor m \rfloor)$	$\lfloor n \rfloor + \lfloor m \rfloor$
`n - m`	$javaSubInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaSubLong(\lfloor n \rfloor, \lfloor m \rfloor)$	$\lfloor n \rfloor - \lfloor m \rfloor$
`n * m`	$javaMulInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaMulLong(\lfloor n \rfloor, \lfloor m \rfloor)$	$\lfloor n \rfloor * \lfloor m \rfloor$
`n / m`	$javaDivInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaDivLong(\lfloor n \rfloor, \lfloor m \rfloor)$	$jdiv(\lfloor n \rfloor, \lfloor m \rfloor)$
`n % m`	$javaModInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaModLong(\lfloor n \rfloor, \lfloor m \rfloor)$	$jmod(\lfloor n \rfloor, \lfloor m \rfloor)$
`(T) n`	$castToByte(\lfloor n \rfloor)$, $castToShort(\lfloor n \rfloor)$, $castToInt(\lfloor n \rfloor)$, $castToLong(\lfloor n \rfloor)$, or $castToChar(\lfloor n \rfloor)$		
`~ n`	$javaBitwiseNegation(\lfloor n \rfloor)$		—
`n << m`	$javaShiftLeftInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaShiftLeftLong(\lfloor n \rfloor, \lfloor m \rfloor)$	—
`n >> m`	$javaShiftRightInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaShiftRightLong(\lfloor n \rfloor, \lfloor m \rfloor)$	—
`n >>> m`	$javaUnsignedShiftRightInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaUnsignedShiftRightLong(\lfloor n \rfloor, \lfloor m \rfloor)$	—
`n & m`	$javaBitwiseAndInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaBitwiseAndLong(\lfloor n \rfloor, \lfloor m \rfloor)$	—
`n | m`	$javaBitwiseOrInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaBitwiseOrLong(\lfloor n \rfloor, \lfloor m \rfloor)$	—
`n ^ m`	$javaBitwiseXOrInt(\lfloor n \rfloor, \lfloor m \rfloor)$	$javaBitwiseXOrLong(\lfloor n \rfloor, \lfloor m \rfloor)$	—

Table A.10 Translation of comprehension expressions (generalized quantifiers)

JML expression e	**JavaDL** $\lfloor e \rfloor \in \mathrm{Trm}_{int}$	
`(\sum` T `x;` n `<= x && x <` m`;` t`)`	$(T')bsum\{int\ x\}(\lfloor n \rfloor, \lfloor m \rfloor, \lfloor t \rfloor)$	$\lfloor T \rfloor = int, t$ of type T'
`(\sum` T `x1, ... xn;` A`;` t`)`	$(T')sum\{T\ x_1\}(\ldots, sum\{T\ x_n\}(\bigwedge_{i=1}^{n} \mathrm{inRange}_T(x_i) \wedge \lfloor A \rfloor, \lfloor t \rfloor)\ldots)$	t of type T'
`(\num_of` T `x;` n `<= x && x <` m`;` B`)`	$(T')bsum\{int\ x\}(\lfloor n \rfloor, \lfloor m \rfloor, \text{if } (\lfloor B \rfloor) \text{ then } (1) \text{ else } (0))$	$\lfloor T \rfloor = int, t$ of type T'
`(\num_of` T `x1, ... xn;` A`;` B`)`	$(T')sum\{T\ x_1\}(\ldots, sum\{T\ x_n\}(\bigwedge_{x=1}^{n} \mathrm{inRange}_T(x_i) \wedge \lfloor A \rfloor, \text{if } (\lfloor B \rfloor) \text{ then } (1) \text{ else } (0))\ldots)$	t of type T'
`(\product` T `x;` n `<= x && x <` m`;` t`)`	$(T')bprod\{int\ x\}(\lfloor n \rfloor, \lfloor m \rfloor, \lfloor t \rfloor)$	$\lfloor T \rfloor = int, t$ of type T'
`(\product` T `x1, ... xn;` A`;` t`)`	$(T')prod\{T\ x_1\}(\ldots, prod\{T\ x_n\}(\bigwedge_{i=1}^{n} \mathrm{inRange}_T(x_i) \wedge \lfloor A \rfloor, \lfloor t \rfloor)\ldots)$	t of type T'
`(\max` T `x1, ... xn;` A`;` t`)`	$max\{T\ x_1\}(\ldots, max\{T\ x_n\}(\bigwedge_{i=1}^{n} \mathrm{inRange}_T(x_i) \wedge \lfloor A \rfloor, \lfloor t \rfloor)\ldots)$	t of type T'
`(\min` T `x1, ... xn;` A`;` t`)`	$min\{T\ x_1\}(\ldots, min\{T\ x_n\}(\bigwedge_{i=1}^{n} \mathrm{inRange}_T(x_i) \wedge \lfloor A \rfloor, \lfloor t \rfloor)\ldots)$	t of type T'

Table A.11 Restrictions on JavaDL types to match JML types

JML Type T'		**JavaDL type** $\lfloor T' \rfloor$	**Restriction formula** $\mathrm{inRange}_{T'}(x)$
`boolean`		*boolean*	*true*
`byte`		*int*	$inByte(x)$
`char`		*int*	$inChar(x)$
`short`		*int*	$inShort(x)$
`int`		*int*	$inInt(x)$
`long`		*int*	$inLong(x)$
`\bigint`		*int*	*true*
`nullable` T	$T \sqsubseteq Object$	T	$select_{boolean}(\mathtt{heap}, x, created) \doteq TRUE \vee x \doteq \mathtt{null}$
`nonnull` T	$T \sqsubseteq Object$	T	$select_{boolean}(\mathtt{heap}, x, created) \doteq TRUE \wedge x \not\doteq \mathtt{null}$
`nonnull` T`[]`^{n}	$T \sqsubseteq Object$	T`[]`^{n}	$select_{boolean}(\mathtt{heap}, x, created) \doteq TRUE \wedge nonNull(\mathtt{heap}, x, n)$
`\locset`		*LocSet*	$disjoint(x, unusedLocs(\mathtt{heap}))$
`\seq`		*Seq*	*true*

Table A.12 Translation of special JML operators of arbitrary type

JML expression e	**JavaDL** $\lfloor e \rfloor \in \mathrm{Trm}_{Any}$
A `?` B `:` C	if $(\lfloor A \rfloor)$ then $(\lfloor B \rfloor)$ else $(\lfloor C \rfloor)$
`\result`	`res`
`\old(`A`)`	$\{\mathtt{heap} := \mathtt{heap}^{pre} \| \mathtt{p}_1 := \mathtt{p}_1^{pre} \| \ldots \| \mathtt{p}_n := \mathtt{p}_n^{pre}\} \lfloor A \rfloor$ (in loop invs.)
`\old(`A`)`	$\{\mathtt{heap} := \mathtt{heap}^{pre}\} \lfloor A \rfloor$ (in method contracts)
`\pre(`A`)`	$\{\mathtt{heap} := \mathtt{heap}^{pre}\} \lfloor A \rfloor$
`(\lblneg` x A`)`	$\lfloor A \rfloor$
`(\lblpos` x A`)`	$\lfloor A \rfloor$
`\dl_func(`$p1$`, ..., `pn`)`	$func(\mathtt{heap}, \lfloor p1 \rfloor, \ldots \lfloor pn \rfloor)$
`(*` *term* `*)`	*term*
`\index`	*index*

Table A.13 Reference expressions

	$e \in$ JExp	$\lfloor e \rfloor \in \mathrm{Trm}_{Any}$
self reference	`this`	`self`
local variable	`v`	`v`
field access	o`.f`	$select_{T'}(\mathtt{heap}, \lfloor o \rfloor, C{::}\mathtt{f})$
static field access	C`.f`	$select_{T'}(\mathtt{heap}, \mathtt{null}, C{::}\mathtt{\$f})$
array access	a`[`i`]`	$select_{T'}(\mathtt{heap}, \lfloor a \rfloor, arr(\lfloor i \rfloor))$
array length	a`.length`	$length(\lfloor a \rfloor)$
pure method	o`.pm(`$p1, \ldots, pn$`)`	$C{::}pm(\mathtt{heap}, \lfloor o \rfloor, \lfloor p1 \rfloor, \ldots, \lfloor pn \rfloor)$
static pure method	C`.pm(`$p1, \ldots, pn$`)`	$C{::}pm(\mathtt{heap}, \mathtt{null}, \lfloor p1 \rfloor, \ldots, \lfloor pn \rfloor)$
model field	o`.mf`	$C{::}mf(\mathtt{heap}, \lfloor o \rfloor)$
static model field	C`.mf`	$C{::}mf(\mathtt{heap}, \mathtt{null})$

A.3 JML Expression Well-Definedness

As explained in Section 8.1.4, in JML, validity depends on the absence of undefinedness. According to the JML reference manual [Leavens et al., 2013], a Boolean expression is satisfied in a state if has the truth value true and "does not cause an exception to be raised." Our translation from JML to JavaDL above ignores this.

KeY can generate well-definedness proof obligations as shown in Section 8.3.3. Table A.16 below gives the full definition of the well-definition operator ω, which provides a formula $\omega(e)$ to every JML expression e, such that e is well-behaving in a state s if and only if $s \vDash \omega(e)$.

[1] These operators may differ in the way well-definedness of expressions is evaluated, see Section A.3 below.

Table A.14 JML location set expressions

JML expression e	**JavaDL** $\lfloor e \rfloor \in \mathrm{Trm}_{LocSet}$
o`.f`	$\{(\lfloor o \rfloor, \mathtt{f})\}$
`\singleton(`o`.f)`	$\{(\lfloor o \rfloor, \mathtt{f})\}$
a`[`i`]`	$\{(\lfloor a \rfloor, arr(\lfloor \mathtt{i} \rfloor))\}$
a`[`i`..`j`]`	$arrayRange(\lfloor a \rfloor, \lfloor i \rfloor, \lfloor j \rfloor)$
a`[*]`	$allFields(\lfloor a \rfloor)$
o`.*`	$allFields(\lfloor o \rfloor)$
`\empty`	$empty$
`\nothing`	$empty$
`\everything`	$setMinus(allLocs, unusedLocs(\mathtt{heap}))$
`\intersect(`s`,`t`)`	$intersect(\lfloor s \rfloor, \lfloor t \rfloor)$
`\set_union(`s`,`t`)`	$union(\lfloor s \rfloor, \lfloor t \rfloor)$
`\set_minus(`s`,`t`)`	$setMinus(\lfloor s \rfloor, \lfloor t \rfloor)$
`\infinite_union(`T `x; `b`; `t`)`	infiniteUnion$\{\lfloor T \rfloor\, x\}($ if $(\mathrm{inRange}_T(x) \wedge \lfloor b \rfloor)$ then $(\lfloor t \rfloor)$ else $(empty)$ $)$
`\reachLocs(f, o, n)`	infiniteUnion$\{Object\ o'\}($ if $(reach(\mathtt{heap}, allObjects(f), \lfloor o \rfloor, o', \lfloor n \rfloor))$ then $(allFields(o'))$ else $(empty)$ $)$
`\reachLocs(f, o)`	infiniteUnion$\{Object\ o'\}($ if $(\exists\ int\ n;\ reach(\mathtt{heap}, allObjects(f), \lfloor o \rfloor, o', n))$ then $(allFields(o'))$ else $(empty)$ $)$

Table A.15 JML sequence expressions

JML expression e	**JavaDL** $\lfloor e \rfloor \in \mathrm{Trm}_{Any}$
`\seq_concat(`$s1$`, `$s2$`)`	$seqConcat(\lfloor s1 \rfloor, \lfloor s2 \rfloor)$
`\seq_empty`	$seqEmpty$
`(`T`)`s`[`i`]`	$seqGet_T(\lfloor s \rfloor, \lfloor i \rfloor)$
s`.length`	$seqLen(\lfloor s \rfloor)$
`\seq_singleton(`e`)`	$seqSingleton(\lfloor e \rfloor)$
s`[`i`..`j`]`	$seqSub(\lfloor s \rfloor, \lfloor i \rfloor, \lfloor j \rfloor)$
`(\seq_def \bigint x; `i`; `j`; `t`)`	$seqDef\{int\ x\}(\lfloor i \rfloor, \lfloor j \rfloor, \lfloor t \rfloor)$
`\values`	$values$

Table A.16 Definition of the well-definedness operator ω

JML expression $e \in$ JExp	**JavaDL formula** $\omega(e) \in$ DLFml	
`x`, `this`, `super`, `null`, `\result`	*true*	for `x` a literal or local variable
$\circ A$	$\omega(A)$	$\circ \in \{$`!`, `-`, `~`$\}$
$A \circ B$	$\omega(A) \wedge \omega(B)$	$\circ \in \{$`==`, `!=`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*`, `&`, `|`, `<==>`, `^`, `>>`, `<<`, `>>>`$\}$
`\`f`(`A`)`	$\omega(A)$	$f \in \{$`fresh`, `new_elems_fresh`$\}$
`\`f`(`A`,` B`)`	$\omega(A) \wedge \omega(B)$	$f \in \{$`set_union`, `intersect`, `set_minus`, `subset`, `disjoint`$\}$
A `/` B, A `%` B	$\omega(A) \wedge \omega(B) \wedge B \not\doteq 0$	
A `&&` B	$\omega(A) \wedge (\lfloor A \rfloor \rightarrow \omega(B))$	
A `==>` B	$\omega(A) \wedge (\lfloor A \rfloor \rightarrow \omega(B))$	
A `||` B	$\omega(A) \wedge (\neg \lfloor A \rfloor \rightarrow \omega(B))$	
A `<==` B	$\omega(A) \wedge (\neg \lfloor A \rfloor \rightarrow \omega(B))$	
A `?` B `:` C	$\omega(A) \wedge (\lfloor A \rfloor \rightarrow \omega(B)) \wedge (\neg \lfloor A \rfloor \rightarrow \omega(C))$	
o`.f`	$\omega(o) \wedge o \not\doteq$ `null`	for an instance field `f`, also if used as location
`C.f`	*true*	for a static field `C.f`, also if used as location
o`.m(`$a_1, \ldots, a_n$`)`	$\omega(o) \wedge o \not\doteq$ `null` $\wedge \bigwedge_{i=1}^{n} \omega(a_i) \wedge pre[a_i/\mathtt{p}_i, o/\mathtt{self}]$	
`new C(`$a_1, \ldots, a_n$`)`	$\bigwedge_{i=1}^{n} \omega(a_i) \wedge pre[a_i/\mathtt{p}_i, o/\mathtt{self}]$	
a`[`i`]`	$\omega(a) \wedge a \not\doteq$ `null` $\wedge\, \omega(i) \wedge 0 \leq i \wedge i < length(a)$	array access, sequence access also if used as location.
a`[`i`..`j`]`	$\omega(a) \wedge a \not\doteq$ `null` $\wedge\, \omega(i) \wedge \omega(j) \wedge 0 \leq i \wedge i \leq j \wedge j < length(a)$	array range (location set)
o`[*]`, o`.*`	$\omega(o) \wedge o \not\doteq$ `null`	
o`.length`	$\omega(o) \wedge o \not\doteq$ `null`	for an array o
`(\`Q T v`;`A`;`B`)`	$\forall \lfloor T \rfloor\, v;\ \big(\omega(A) \wedge (\lfloor A \rfloor \rightarrow \omega(B))\big)$	$Q \in \{$`forall`, `exists`, `min`, `max` `infinte_union`, `sum`, `product`$\}$
`(\seq_def` T v`;`A`;`B`;`C`)`	$\forall$`int` $v; (\lfloor A \rfloor \leq v < \lfloor B \rfloor \rightarrow \omega(C)) \wedge \omega(A) \wedge \omega(B)$	$T \in \{$`int`, `\bigint`$\}$
`(`T`)`t	$\omega(t) \wedge instance_{\lfloor T \rfloor}(\lfloor t \rfloor) \doteq TRUE$	
t `instanceof` T	$\omega(t)$	
`\old(`A`)`	$\{\mathtt{heap} := \mathtt{heap}^{pre}\}\omega(A)$	
`\invariant_for(`A`)`	$\omega(A) \wedge A \not\doteq$ `null`	