A Translation from JML to JavaDL

Studienarbeit
Christian Engel

Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Verantwortlicher Betreuer: Prof. Dr. Peter H. Schmitt
Betreuer: Andreas Roth

February 21, 2005
Danksagung
Ich möchte an dieser Stelle meinen Betreuern Prof. Dr. Peter H. Schmitt, an dessen Lehrstuhl diese Studienarbeit entstand, und Andreas Roth, der sich stets für das Diskutieren offener Fragen die Studienarbeit betreffend Zeit nahm, für ihre Unterstützung beim Erstellen dieser Arbeit danken.

Erklärung
Ich erkläre hiermit, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Hilfsmittel verwendet zu haben.

Christian Engel
Karlsruhe, February 21, 2005
## Contents

1 Introduction  
   1.1 The KeY Project ........................................... 1  
      1.1.1 JavaDL .................................................. 1  
   1.2 The Java Modeling Language (JML) ........................ 3  
   1.3 Goal of this Work ........................................... 4  
   1.4 Quality and Correctness Criteria ........................... 4  
   1.5 Structure ................................................... 4  

2 Design Decisions ............................................... 6  
   2.1 Syntactical Correctness of Specifications .................. 6  
      2.1.1 Specifications and Privacy Modifiers ................... 6  
   2.2 Boolean Expressions ......................................... 6  
   2.3 Queries ................................................... 7  
   2.4 Errors ................................................... 7  
   2.5 Redundant Specifications ............................... 8  

3 JML Specifications and their Translation to JavaDL ............ 10  
   3.1 Method Specifications ...................................... 11  
      3.1.1 Lightweight specification cases ..................... 12  
      3.1.2 Heavyweight specification cases ..................... 12  
      3.1.3 Nested Method Specifications ......................... 13  
      3.1.4 Clauses ............................................... 14  
      3.1.5 Pure Methods and Constructors ......................... 18  
      3.1.6 Helper Methods and Constructors ...................... 19  
      3.1.7 Inheritance of Method Specifications .................. 19  
   3.2 Class and Interface Specifications ......................... 19  
      3.2.1 Invariants .............................................. 20  
      3.2.2 History Constraints ................................... 21  
   3.3 Model Methods ............................................ 21  
   3.4 Model Fields .............................................. 22  
      3.4.1 The Represents Clause ................................ 22  
      3.4.2 Translation of the Represents Clause ................. 23  
      3.4.3 Inheritance of Model Fields ......................... 25  
   3.5 Types .................................................. 27
A Syntax of KeYJML

A.1 Field Syntax .................................................. 57
A.2 Behavioral Specification Syntax for Types .................... 58
A.3 Store Ref Syntax .................................................. 58
A.4 Behavioral Specification Syntax for Methods .................. 58
  A.4.1 Lightweight Specification Cases .......................... 59
  A.4.2 Heavyweight Specification Cases ......................... 59
  A.4.3 Method Specification Clause Syntax ....................... 59
A.5 Predicate and Specification Expression Syntax ............... 60
A.6 Expression Syntax .............................................. 60
A.7 Tokens .......................................................... 61
Chapter 1

Introduction

1.1 The KeY Project

Goal of the KeY Project [Key00] is to integrate formal methods in the industrial software engineering process. The KeY System, a semi-automatic proof system is therefore not only available as standalone prover, but also as a plugin for a commercial CASE tool, the Together Control Center. TogetherCC uses UML/OCL which is translated (see [Ke02]) to JavaDL (see [Bec01]), a dynamic logic for JavaCard, on which the verification process within the KeY System is based.

JavaCard is a subset of Java tailored to the usage on smartcards. It does not contain among other things concurrency and floating point arithmetics which makes verification of JavaCard programs a lot easier than it would be the case for Java.

It should be mentioned on that occasion that the KeY system is not tied to JavaDL, but is easily adaptable, mainly due to the flexibility of the taclet mechanism, to other languages, as the support for Abstract State Machines and the recently done integration of ODL (see [Pa04]) show.

1.1.1 JavaDL

The target language for the JML translation is JavaDL. This section gives a short introduction to JavaDL and point out some aspects that are relevant for this work. For a deeper insight in JavaDL see [Bec01].

Syntax

Program Contexts Syntax and semantics of JavaDL are defined with respect to the context, which can be provided by any executable JavaCard program. Large parts of the signature such as the types of classes contained in the context or methods and fields which are represented in JavaDL as (nonrigid) functions are derived from the context.

Types The types available in JavaDL that are relevant for this work are:

- The primitive types of JavaCard.
CHAPTER 1. INTRODUCTION

- Class types derived from the context.
- Array types for each JavaDL type including each array type.

**Rigid and Non-Rigid Terms**  JavaDL distinguished between rigid and non-rigid terms. The result of the evaluation of a rigid term doesn’t depend on the state it is evaluated in whereas for a non-rigid term the state has to be taken into account.

It is obvious that the value of a (non-final) field or a local variable declared somewhere in a program surely depends on the state it is evaluated in. Therefore these java entities have to be represented by non-rigid terms in JavaDL.

**Variables**  In JavaDL there are two kinds of variables, *program variables* and *logic variables*.

Program variables can represent local variables in Java programs or fields of classes. They are no variables from a predicate logic point of view. In JavaDL program variables are non-rigid constants and can consequently not be quantified.

Logic variables are variables in the original meaning. They can only appear quantified within JavaDL formulas and are rigid.

**Functions**  As for variables there are rigid and non-rigid functions.

As already mentioned program variables are non-rigid functions of arity 0. Attributes and arrays are also interpreted as non-rigid functions \( (a[i] \equiv [[a, i]) \text{ and } o.\text{attr} \equiv \text{attr(o)} \)\).

An important kind of non-rigid function are the so called queries which are side-effect-free Java methods that occur outside a modality in the logic. Queries will be used in the JML translation on several occasions, for instance for translating method calls that occur within a specification or as a means to translate model fields in a more readable way.

**Updates**  Updates are a syntactical construct that describes the state in which the formula it is attached to should be evaluated, which means \( \phi^{p \leftarrow x} \) is equivalent to \( \langle p = x; \rangle \phi \). \( p \) and \( x \) are not allowed to have any side effects in this case.

Updates will appear in two different notations:

- as exponents: \( \phi^{p \leftarrow x} \)
- or \( \{ p := x \} \phi \)

There is semantically no difference between the two notations and usually the first one is used. Nevertheless in examples showing generated proof obligations you will find the second variant since in these cases the shown formula should closely resemble its appearance in the prover.

**Method Body Statement**  In the following sections we will in some examples encounter *method body statements*. They represent an intermediate stage in the evaluation of a methodcall. The expression:

\[
\text{SomeType(result)} :: \text{obj.method(p1, \ldots ,pn)};
\]
1.2. THE JAVA MODELING LANGUAGE (JML)

for instance means:

- the method body of method is implemented in SomeType
- obj is either the prefix of method or the context method is evaluated in, if we have no prefix.
- the method obj.method is called with the parameters p1, ..., pn
- the result of the method call is assigned to the location result
- obj, p1, ..., pn and result are side-effect-free.

Formulas

- Every definable predicate logical formula over the signature of JavaDL is a JavaDL formula.
- For every JavaDL formula \( \phi \) and every program \( p \), \( \langle p \rangle \phi \) and \( [p] \phi \) are JavaDL formulas.
- Every formula created from a JavaDL formula by instantiation of a subformula with a JavaDL formula is a JavaDL formula.

Semantics

The underlying concept for the interpretation of JavaDL formulas is as for DL in general the concept of Kripke structures. A Kripke structure is given by the tuple \( K = (S, \rho) \) where \( S \) is the set of predicate logic structures definable for a given context and \( \rho \) is a function that assigns to a program its corresponding transition relation (which is a partial function to be precise, as \( K \) is deterministic) over \( S \): \( \rho : P \rightarrow 2^{S \times S} \), where \( P \) is the set of valid JavaCard programs definable for a given context. All \( s \in S \) have the same universe.

Example 1.1.1 Let \( s \) be a predicate logic structure with \( s \in S \), \( \langle p \rangle \phi \) JavaDL formula and \( I_s \) the interpretation in the predicate logic structure \( s \):

\[
I_s(\langle p \rangle \phi) = \begin{cases} 
I_{\rho(p)(s)}(\phi), & \text{if } p \text{ terminates} \\
false, & \text{otherwise}
\end{cases}
\]

Example 1.1.2 (Hoare Triple) A Hoare Triple \([\phi][p][\psi]\), where \( p \) is a program and \( \phi \) and \( \psi \) are formulas in predicate logic, can be expressed as:

\[ \phi \rightarrow \langle p \rangle \psi \]

1.2 The Java Modeling Language (JML)

JML [LBR03] is a behavioral interface specification language tailored to Java. Its syntax is based on the Java syntax enriched by quantified expressions and other constructs.

JML specifications are attached to the code in the form of comments. This and the fact that JML’s syntax is very Java-like makes it easier to use than OCL for Java programmers without experiences in formal methods.
Besides the possibility to specify pre-, postconditions, invariants and history constraints, which are essential constructs for doing design by contract (DBC) [CLSE04], JML provides mechanisms to describe concurrency and realtime aspects\(^1\).

1.3 Goal of this Work

Goal of this work is to design and implement a translation from JML to JavaDL. This provides an additional input language for the KeY System.

As integral part of this minor thesis the translation of the subset of JML described in this work, is integrated into the KeY System.

1.4 Quality and Correctness Criteria

Some important quality criteria for the translation of JML expressions and whole specifications are:

- The proof obligation created for a specification should closely resemble the structure of the specification in order to make it easier to reproduce for the user what is actually expressed by this proof obligation.

- Besides having a form that is convenient for the user, machine usability of the created formulas should also be a quality aspect of the translation, which means the structure of the formulas serving as proof obligations should aim at requiring as less user interaction as possible and at leading to potentially short proofs.

The correctness considerations for the translation are based on the JML semantics described in [Ref04] and [LBR03]. Given the absence of an "official" formal semantics, correctness means in this context matching the prosaic descriptions provided by the mentioned references as closely as possible.

Thus this work provides a formal semantics for JML described in JavaDL.

1.5 Structure

Chapter 2 describes some fundamental design decisions for the translation of JML specification and especially expressions.

Chapter 3 outlines the JML subset (KeYJML) covered by this work and its translation to JavaDL. The focus of this chapter is directed at the different kinds of specification clauses and basic concepts JML offers.

Chapter 4 gives an overview of the translation of JML expressions to JavaDL terms. The scope of JML described in this and the previous chapter corresponds to the scope actually implemented in the KeY-system.

Chapter 5 explains the different kinds of proof obligations generated from the specifications.

\(^{1}\)This two issues are not discussed in this work.
Chapter 6 shows some implementational details.

Chapter 7 gives a short overview of related works and tools.

Chapter 8 combines a summary of this work with an outlook to possible extensions and future work.
2.1 Syntactical Correctness of Specifications

It is assumed that the JML specification serving as input for the translation are syntactically valid. Thus some aspects such as the correct (specification) visibility of fields used in JML specifications are not checked by the JML to JavaDL translation.

2.1.1 Specifications and Privacy Modifiers

Method specifications, invariants and constraints can be declared with the privacy modifiers public, protected and private (see [Ref04, p.11]). They determine which fields and methods can be used in it, namely those who have at least\(^1\) the visibility of the specification, but have no effects on the semantics of this specification.

Since we assume that the JML specifications we get are syntactically valid, which means from a pragmatic point of view parsable with the official JML parser, no further checks concerning this issue are performed during the translation.

2.2 Boolean Expressions

JML expressions are written in Java syntax. Thus pre- and postconditions (among other things), which would be represented as formulas in JavaDL, are expressions of type boolean in JML.

In order to not be bothered with too many case distinctions, whether a boolean expression occurs for instance on the top level of a specification clause, where there is no alternative to translating it to a formula, or in a context, where it would be possible to translate it to a boolean term in JavaDL, boolean expressions are always translated to formulas.

This makes further transformation steps necessary, if a boolean expression occurs in a context, where a formula is syntactically not possible. In these cases a boolean expression is translated to a query with a suitable specification. Details on this issue can be found in section 4.14.8.

\(^1\)for the underlying order private \(\leq\) default access \(\leq\) protected \(\leq\) public
2.3 Queries

JML allows method calls of side effect-free methods\textsuperscript{2} within a specification. The JavaDL equivalents are called queries, which are non-rigid function corresponding to side effect-free Java methods.

Besides for the translation of these method calls, queries are used for several other purposes in this work, such as the translation of conditional expressions, generalized quantifiers or the already mentioned arguments of type boolean. The semantics of a newly introduced query\textsuperscript{3} is always provided by a suitable specification, created from the expression it replaces. It never has an implementation.

The advantage of using queries as substitutes for certain kinds of expressions is basically that this leads to a shorter formula, whose structure is simpler and better resembles the original structure of the specification it was derived from.

One disadvantage is that the resolution of queries leads to slightly longer proofs, which is acceptable as the additional proof steps usually require no user interaction.

2.4 Errors

A \texttt{java.lang.Error} indicates similar to a \texttt{java.lang.Exception}, a malfunction of a piece of Java code. But unlike an \texttt{Exception} it is usually not thrown by this code itself, but by the \texttt{Java Virtual Machine} because of some "external" reasons. Here is a short excerpt from the \textit{Java API Documentation}:

An \texttt{Error} is a subclass of \texttt{Throwable} that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The \texttt{ThreadDeath} error, though a "normal" condition, is also a subclass of \texttt{Error} because most applications should not try to catch it.

A method is not required to declare in its throws clause any subclasses of \texttt{Error} that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur.

Because of this special nature of \texttt{Errors} one can neither seriously demand that a method does not throw an \texttt{Error} nor specify any conditions (different from \texttt{true}) that must hold after an \texttt{Error} has been thrown. Thus JML always allows methods to throw a \texttt{java.lang.Error} regardless of their specification.

But what does this mean for proving a JML specification or using it as a basis for a lemma rule for the specified method?

To answer this question let us first have a look at how JavaDL in general deals with errors.

Let \( \langle \text{Object } a = \text{null;} \ a = \text{new Object(); } a \neq \text{null} \rangle \) be a JavaDL formula. Then it is derived in the following (shortened and without going into any detail

\textsuperscript{2}Those methods are declared with the JML modifier pure.

\textsuperscript{3}That is a query that is not the result of a translated method call.
of object creation) way:

\[
\begin{align*}
\text{true} & \vdash \{a := \text{null}\}(a = c); a \neq \text{null} \\
\text{true, false} & \vdash \{a := \text{null}\}(a = c); a \neq \text{null}, \{a := \text{null}\}(\text{throw new OOMError}()); a \neq \text{null} \\
& \vdash \{\text{Object a = null; a = new Object}(); a \neq \text{null} \}
\end{align*}
\]

where \(c\) is a constant representing the newly created Object.

The creation of an Object could for instance cause an `OutOfMemoryError`. As this example shows us this type of error is simply ignored. This is due to the fact that JavaDL abstracts from the limitations of a finite physical memory. If that would not be the case a derivation including the possibility of errors would look like this:

\[
\begin{align*}
\text{true} & \vdash \{a := \text{null}\}(a = c); a \neq \text{null} \\
\text{true, false} & \vdash \{a := \text{null}\}(a = c); a \neq \text{null}, \{a := \text{null}\}(\text{throw new OOMError}()); a \neq \text{null} \\
& \vdash \{\text{Object a = null; a = new Object}(); a \neq \text{null} \}
\end{align*}
\]

Where OOMError stands for OutOfMemoryError.

Ignoring errors thrown by the JVM the way it is done by JavaDL makes quite sense, because otherwise we would have to deal with indeterministic Kripke Structures which is surely not what we want.

The JML translation follows the JavaDL approach and ignores errors which are thrown by the Java Virtual Machine. A question that remains is how to deal with errors explicitly thrown by the specified Java code, which is a tricky issue as for instance the method

```java
public int m(){
    throw new Error();
}
```

will meet every possible specification according to the JML semantics, which is surely not the primary intention of JML for always allowing a method to throw an error. This problem will be picked up again in section 3.1.4, where we will see that it is not possible with KeYJML to misuse errors as seen above to make a method meet an arbitrary specification.

### 2.5 Redundant Specifications

Several JML specification clauses can be marked with the suffix `.redundantly` in order to indicate that they are expressing a redundant specification (declaring a precondition `redundantly expr`; for instance means that `expr` is implied by other preconditions of the enclosing specification case).

We consider this to be a useful construct, which primarily serves as an annotation for the user to make specifications more understandable but which

---

4 An error inheriting `java.lang.Error` should usually not be thrown by the Java code itself, but nevertheless it is possible and sometimes in cases of a heavy malfunction it could even be sensible, so we can’t ignore this possibility.
2.5. REDUNDANT SPECIFICATIONS

is of no interest for our purposes. Hence the JML to JavaDL translation will not distinguish between redundant and nonredundant specifications or in other words: All specifications are considered to be nonredundant regardless of being declared as redundant or not. There will also be no possibility provided to prove whether the redundant specification is really implied by other specifications or not.
Chapter 3

JML Specifications and their Translation to JavaDL

This chapter describes the subset (KeYJML) of JML actually covered by the JML to JavaDL translation. Thus it will not give a complete overview of all the features of JML. For a complete account on JML refer to [Ref04].

KeYJML contains

- basically all Java expressions allowed in JML specifications.
- several special JML constructs such as quantified expressions.
- JML clauses for invariants and history constraints.
- JML clauses for the specification of pre- and postconditions, exceptional postconditions, modifier sets and nontermination aspects of methods.
- the different types of method specification cases JML distinguishes (normal behavior, exceptional behavior, ...)
- support for model fields and model methods.

KeYJML does currently not contain:

- datagroups
- special JML expressions for arrays and attributes as used in assignable clauses.
- loop invariants
- \asserts
- \fresh
- \reach
- \invariant_for
- \working_space
3.1 Method Specifications

Method specifications ([Ref04, p.36]) are attached to method declarations as comments. They can specify pre- and postconditions of methods (via requires and ensures clauses), postconditions for the case that the termination was caused by a thrown exception (signals clause), preconditions under which the method won’t return to its caller (diverges clause) and the assignable or modifies clause which lists all locations that may be modified by the method.

Syntax:

\[
\text{<method-specification> ::= <specification> } \\
\text{ | <extending-specification>}
\]

\[
\text{<specification> ::= <spec-case-seq>}
\]

\[
\text{<spec-case-seq> ::= <spec-case> [ also <spec-case-seq> ]}
\]

\[
\text{<spec-case> ::= <lightweight-spec-case> | <heavyweight-spec-case>}
\]

\[
\text{<extending-specification> ::= also <specification>}
\]

There are basically two kinds of method specfications: lightweight and heavyweight specification. One can consider lightweight specifications as a kind of incomplete specification as the default values in table 3.1 also show. Another difference is that the visibility of a heavyweight specification can be defined using the privacy modifiers known from Java, whereas lightweight specifications inherit the visibility from the specified method.

<table>
<thead>
<tr>
<th>clause</th>
<th>lightweight</th>
<th>heavyweight</th>
</tr>
</thead>
<tbody>
<tr>
<td>requires</td>
<td>\text{not_specified}</td>
<td>\text{true}</td>
</tr>
<tr>
<td>diverges</td>
<td>\text{not_specified}</td>
<td>\text{false}</td>
</tr>
<tr>
<td>assignable</td>
<td>\text{not_specified}</td>
<td>\text{everything}</td>
</tr>
<tr>
<td>ensures</td>
<td>\text{not_specified}</td>
<td>\text{true}</td>
</tr>
<tr>
<td>signals</td>
<td>(Exception) \text{not_specified}</td>
<td>(Exception) \text{true}</td>
</tr>
</tbody>
</table>

Table 3.1: Default values for method specification cases

This distinction JML makes between lightweight and behavior specification cases isn’t exactly reflected in the translation. As we have to assume some
"usable" default values also for lightweight specifications, we use the same as for
a heavyweight specification, as these are of course just the ones that intuitively
make sense\(^1\). The visibility of a specification is ignored anyway (see section
2.1.1). After all the translation makes no difference between these two kinds of
specification cases.

### 3.1.1 Lightweight specification cases

Syntax:

```xml
<lightweight-spec-case> ::= <generic-spec-case>
<generic-spec-case> ::= <spec-header> [ <generic-spec-body> ]
<spec-header> ::= (requires-clause)+
<generic-spec-body> ::= <simple-spec-body>
| '{|' <generic-spec-case-seq> '|}'
<generic-spec-case-seq> ::= <generic-spec-case> (also <generic-spec-case>)*
<simple-spec-body> ::= (simple-spec-body-clause)+
| '{|' <simple-spec-body-clause-seq> '|}'
<simple-spec-body-clause> ::= <ensures-clause> | <signals-clause>
| <assignable-clause> | <diverges-clause>
```

Lightweight specification cases are treated like \textit{generic behavior speccases},
which are discussed in the next section. For omitted clauses or clauses specified with
\texttt{not_specified} the default value for heavyweight specifications is taken.

### 3.1.2 Heavyweight specification cases

Syntax:

```xml
<heavyweight-spec-case> ::= [ <privacy> ] behavior <generic-spec-case>
| [ <privacy> ] exceptional_behavior <exceptional-spec-case>
| [ <privacy> ] normal_behavior <normal-spec-case>
<privacy> ::= public | protected | private
<exceptional-spec-case> ::= <spec-header>
| [ <exceptional-spec-body> ]
| <exceptional-spec-body>
<exceptional-spec-body> ::= (exceptional-spec-clause)+
| '{|' <exceptional-spec-case-seq> '|}'
<exceptional-spec-clause> ::= <diverges-clause> | <assignable-clause>
| <signals-clause>
<exceptional-spec-case-seq> ::= <exceptional-spec-case> (also <exceptional-spec-case>)*
<normal-spec-case> ::= <spec-header> [ <normal-spec-body> ]
| <normal-spec-body>
<normal-spec-body> ::= (normal-spec-clause)+
| '{|' <normal-spec-case-seq> '|}'
```

\(^1\)We usually want a method to terminate (\texttt{diverges false}). If we don’t say anything
about the other clauses the default value should be the least restrictive value possible
which is \texttt{true} for the pre- and postcondition, \texttt{false} for the assignable clause and
\texttt{Exception} true.
3.1. METHOD SPECIFICATIONS

<normal-spec-clause> ::= <diverges-clause> | <assignable-clause>
| <ensures-clause>

<normal-spec-case-seq> ::= <normal-spec-case>
(also <normal-spec-case>)+

Heavyweight specification cases are again subdivided into three different types of specification cases, namely

- behavior or generic specification cases (which are the ones considered by the translation to be equivalent to lightweight specifications)
- normal behavior specification cases
- exceptional behavior specification cases

where the last two are in the end just refinements of a behavior specification case. A normal behavior specification case (which starts with the keyword normal_behavior) is just syntactic sugar for a behavior specification containing the signals clause signals (Exception) false; and thus not allowing the specified method to terminate by throwing an exception. An exceptional behavior specification case (starting with the keyword exceptional_behavior) on the other hand is just syntactic sugar for a behavior specification which contains the ensures clause ensures false; which means it is not allowing the specified method to terminate normally.

As one can clearly see it is not necessary for the translation to distinguish between these three types of method specification. Nevertheless this proves in some cases quite useful as it allows easily realizable optimizations for both proof obligations\(^2\) and lemmas generated from these specifications.

3.1.3 Nested Method Specifications

A nested method specification of the form:

```
[specholder]
{|
  speccase1
  also
  speccase2
  also
  ...
  also
  speccasen
|}
```

is syntactic sugar for the speccase sequence:

```
specholder
speccase1
also
specholder
speccase2
```

\(^2\)the proof obligations, which are generated for proving the specification, are meant here.
CHAPTER 3. JML SPECIFICATIONS AND THEIR TRANSLATION TO JAVADL

Also

Also

specheader

speccasen

and is translated accordingly.

3.1.4 Clauses

Requires and Ensures clauses

The Requires clause is used for specifying the precondition of a method, the Ensures clause for specifying the postcondition that holds after a normal termination of the method when it was invoked in a state in which the precondition specified by the corresponding Requires clause and all applicable invariants held. The syntax of the Requires and the Ensures clause is as follows:

<requires-clause> ::= requires (<expression> | \not_specified);
<ensures-clause>  ::= ensures (<expression> | \not_specified);

where <expression> is an expression of type boolean. In the case of an omitted requires or ensures clause or \not_specified we use true as default (see table 3.1).

Writing several requires/ensures clauses within one specification case is equivalent to writing the conjunction of the expressions contained in these clauses in a single requires/ensures clause. Thus it is sufficient to consider w.l.o.g. a single requires and a single ensures clause.

Diverges

Diverges clauses can be used to specify non-termination aspects of methods.

Syntax:

<diverges-clause> ::= <diverges-keyword> <pred-or-not> ;
<diverges-keyword> ::= diverges | diverges_redundantly

The diverges clause specifies the precondition under which the method may not return (either normally or by throwing an exception) to its caller.

Not returning to its caller can mean in this context infinite looping or calling System.exit() as explained in [Ref04, p.59] whereas the throwing of a java.lang.Exception would be considered to be a return to the caller (see the paragraph about signals for more about exceptions).

There are a number of other papers ([LBR03], [CLSE04] for example) which mention that cases in which an java.lang.Error is thrown should also be specified in the diverges precondition. This is no real contradiction to [Ref04] as situations like an infinite recursion which lead theoretically to nontermination, cause in practice for instance a StackOverflowError (as in this case) because of technical restrictions such as a finite amount of memory and so on.

As errors in contrast to exceptions should usually not be caught by the caller of a method (see section 2.4) it would make sense to classify the throwing of an
error as a situation in which a method does not return to its caller. Thus if a method can throw errors under certain preconditions, these preconditions have to be covered by the condition(s) specified by the diverges clause(s). This only refers to those errors thrown by an explicit `throw` statement in the code and not to those thrown by the `Java Virtual Machine`\(^3\).

JML offers the possibility to annotate methods with several `diverges` clauses which is equivalent to writing the disjunction of them in a single clause. So it is sufficient if we consider methodspecs with at most one `diverges` clause. With `diverges_redundantly` one can point out that a `diverges` clause is redundant, but this has no effect on the translation of this clause.

**Signals**

`signals` clauses specify the exceptional behavior of methods.

**Syntax:**

```plaintext
<signals-clause> ::= <signals-keyword> '(' <reference-type> ')'
<pred-or-not> ;
<signals-keyword> ::= signals | exsures
```

The `signals` clause specifies which condition must hold after an exception has been thrown. `signals` clauses can only be used for classes that inherit from `java.lang.Exception`, consequently also classes extending `java.lang.Error` (see also paragraphs 3.1.4 and 2.4) are not allowed to be the "parameter" of a `signals` clause.

As for the other clauses it is possible to transform several `signals` clauses into a single `signals` clause:

```plaintext
signals (E1) R1;
signals (E2) R2;
```

is equivalent to:

```plaintext
signals (Exception e) ((e instanceof E1) ==> R1)
&& ((e instanceof E2) ==> R2);
```

The translation of specifications with multiple `signals` clauses is done accordingly.

**Example 3.1.1** This is an Example for a behavioral methodspecification with `signals` clauses:

```plaintext
/*@ public behavior
@ ensures \result == elements[i];
@ signals (NullPointerException) elements == null;
@ signals (ArrayIndexOutOfBoundsException) elements != null
@ && (i<0 || i>elements.length);
@*/
public Object getElementAt(int i){
    return elements[i];
}
```

\(^3\)These errors are ignored as explained in section 2.4, because this would introduce an indeterminism to the Java semantics
This specification is equivalent to its following desugared version which takes into account the defaults for omitted clauses:

```java
/*@ public behavior
  @ requires true;
  @ assignable \everything;
  @ diverges false;
  @ ensures \result == elements[i];
  @ signals (NullPointerException) elements == null;
  @ signals (ArrayIndexOutOfBoundsException) elements ![null]
    @ & (i<0 || i>elements.length);
  @*/
public Object getElementAt(int i){
  return elements[i];
}
```

The meaning of this specification is as follows:

- If the method terminates normally then the returned Object is `elements[i]`.
- If a NullPointerException is thrown then `elements == null` holds in the poststate.
- If an ArrayIndexOutOfBoundsException then `elements != null & &
  (i < 0 || i >= elements.length)` holds in the poststate.

This specification can be expressed in JavaDL for example in the following way:

```
{old_i := i}(self != null =>
  \{ java.lang.Exception exc = null;
    try {
      Test(result)::self.getElementAt(i);
    } catch (java.lang.Exception e) {
      exc = e;
    }
  \} (exc == null => result = self.elements[old_i]
    \ & (exc != null
      \rightarrow ((exc instanceof NullPointerException) = TRUE
        \rightarrow self.elements = null)
    \ & (exc instanceof ArrayIndexOutOfBoundsException) = TRUE
      \rightarrow self.elements != null
        \ & (old_i < 0
          \ | old_i >= self.elements.length)))
```

The program variable `old_i` is introduced because parameters in JML-specifications are always evaluated in the prestate. So it would be equivalent to write `\old(p)` instead of `p` whenever a parameter `p` occurs in the method specification. The translation of the `\old` construct is described in section 4.14.1.

`self != null`, where `self` has the type of the class the specified method is contained in, is added to the precondition of every non-static method.
Example 3.1.2 Instead of using a single specification case for this method, one could also divide the specification into one specification for the normal and one for the exceptional behavior, which could for instance look like that\(^4\):

```java
/*@ public normal_behavior
  @ requires elements != null && 0<=i && i<elements.length;
  @ ensures result == elements[i];
  @ public exceptional_behavior
  @ requires !(elements != null && 0<=i && i<elements.length);
  @ signals (NullPointerException) elements == null;
  @ signals (ArrayIndexOutOfBoundsException) elements != null
    @ && (i<0 || i>=elements.length);
  @*/

public Object getElementAt(int i){
    return elements[i];
}
```

The generated proof obligations of the normal\_behavior specification looks like that:

\[
\{\text{old}_i := i\} \text{( self \neq null } \land \text{ self.elements } \neq \text{ null } \land 0 \leq i \land i < \text{ self.elements.length}) \rightarrow \text{ Test(result)::self.getElementById(i); result = self.elements[old_i]}\]

The generation of proof obligations for JML-specifications is described in section 5.

Assignable

Syntax:

\[
<\text{assignable-clause}> ::= <\text{assignable-keyword}><\text{conditional-store-ref-list}> \ ;
<\text{assignable-keyword}> ::= \text{ assignable } | \text{ modifiable } | \text{ modifies }
<\text{conditional-store-ref-list}> ::= <\text{conditional-store-ref}> (, <\text{conditional-store-ref}> )*
<\text{conditional-store-ref}> ::= <\text{store-ref}> [ if predicate ]
<\text{store-ref-list}> ::= <\text{store-ref}> (, <\text{store-ref}> )*
<\text{store-ref}> ::= <\text{store-ref-expression}>
| <\text{store-ref-keyword}>
<\text{store-ref-expression}> ::= <\text{store-ref-name}>
| <\text{store-ref-name-suffix}> 
<\text{store-ref-name}> ::= ident | this
\]

\(^4\)Just in order to give no wrong impression: This specification is not equivalent to the specification in the previous example, as the behavior spec case in example 3.1.1 would also allow the method to throw Exceptions if it is called in a state that doesn’t imply the precondition of the exceptional behavior spec case in 3.1.2.
Assignable or modifies clauses are the JML equivalent to the modifier sets discussed in [BS03]. The assignable clause contains all the locations (also model fields are permitted) the method is allowed to change, which means that the set of locations actually changed by the method must always be a subset of the set of locations specified by the assignable clause, but there is no obligation for the assignable clause to be minimal. Thus \texttt{\texttt{everything}}, which indicates that there are no restrictions of what locations the method may assign to, is always a valid assignable set.

Local variables of the method including parameters can always be assigned to and thus are not permitted to occur in an assignable clause. This also applies to nonstatic fields of objects created during the execution of the method. This includes that a constructor may always assign to nonstatic fields of the object it creates\footnote{This applies of course only to those cases when the method is called in a state, that implies the precondition of the specification case the assignable clause belongs to.} irrespective of the assignable clause.

Expressing that a method won’t change any location (for a certain spec case) can be achieved by using the keyword \texttt{\texttt{nothing}}. Declaring a method pure has the same effect but for all its specifications.

As it is emphasized in [Ka03], we can only use specifications for eliminating method calls, if they have a finite modifier set. Thus only those specification cases, that meet this requirement, are used for creating rules for that.

For this purpose we have to replace the model fields in the assignable clause by the set of locations that can actually change if the value of the model field changes. JML calls these sets datagroups [LBR03, p.24]. The KeY-JML translation can’t handle datagroups at the moment, but they will be needed to have a neat support for model fields in assignable clauses.

\textbf{Remark 3.1.3 (Implicit Fields)} Object creation is simulated within KeY by assignments to implicit attributes such as \texttt{<created>}, which are implicitly defined for every object type. These implicit attributes cannot be mentioned in assignable clauses. In addition \texttt{\texttt{nothing}} and other JML constructs used to describe assignable modes do not refer to implicit fields. Thus implicit fields are always allowed to change.

\subsection*{3.1.5 Pure Methods and Constructors}

Pure methods/constructors (see also [Ref04, p.32]) are declared with the JML modifier pure\footnote{One might object here that a constructor doesn’t really create the object but only initializes it after it has been allocated by the \texttt{new} operator. Nevertheless this is the semantics for assignable clauses and constructors provided by JML and section 3.1.5 will show that this offers also some advantages.} (as usual written in comments: /*@pure@*/). The main purpose of pure for us is to indicate whether a method can be used in a specification or not. Therefore pure methods have to meet the following requirements:
3.2. CLASS AND INTERFACE SPECIFICATIONS

- no sideeffects\(^7\) (assignable \nothing;)
- termination (diverges clause omitted or diverges false;)

Note that a pure constructor may assign to nonstatic fields of the object it creates which is consistent to the semantics of the assignable clause. Thus the purity of a constructor \(C(T_1 p_1, \ldots, T_n p_n)\) means intuitively that the expression \(\text{new } C(x_1, \ldots, x_n)\) is sideeffect free (if and only if \(x_1, \ldots, x_n\) are sideeffect free) but not that for instance the statement \(\text{this}(x_1, \ldots, x_n)\) is sideeffect free.

Choosing the semantics of the assignable clause this way, makes it possible to use constructors within specifications, which otherwise would only have been possible for default constructors that do not initialize any location.

Checking the purity of a method automatically can be done by creating a specification of the form:

```java
/*@ public behavior
   @ assignable \nothing;
   @*/
```

which then can be proven by using KeY.

In contrast to JML, where model methods can be used in model programs, our only use for model methods is within specifications. That’s why our model methods are virtually obliged to be declared pure.

3.1.6 Helper Methods and Constructors

Helper methods/constructors\(^8\) are methods/constructors that neither can expect invariants to hold in the prestate nor have to preserve invariant or fulfill constraints. This finds expression in the proofobligations and lemmas for helper methods of course, namely to that effect that the invariants and constraints are not added to the pre- and postcondition (see also section 5).

3.1.7 Inheritance of Method Specifications

Methods overriding a method from a supertype inherit its specification, which enforces behavioral subtyping (see [DL95]).

3.2 Class and Interface Specifications

Syntax:

```
<jml-declaration> ::= <modifiers> <invariant>
   | <modifiers> <history-constraint>
   | <modifiers> <represents-decl>
```

JML provides several possibilities to express properties of types such as invariants, which must hold in every state that is visible\(^9\) or history constraints (to

---

\(^7\)From a JavaDL point of view pure methods aren’t guaranteed to be sideeffect free, since even the creation of an object is a sideeffect, because it changes some implicit fields of the affected type.

\(^8\)Methods/constructors declared with the JML modifier helper

\(^9\)Before/after the execution of methods and constructors or after static initialization
which we refer to as *constraints* from now on), which can describe the way certain values are changed by methods. The *represents* clause specifies how a *model field* is represented by the concrete implementation. More about *model fields* and *represents* can be found in section 3.4.

### 3.2.1 Invariants

**Syntax:**

```
<invariant> ::= <invariant-keyword> <predicate> ;
<invariant-keyword> ::= invariant | invariant_redundantly
```

JML distinguishes between static and instance invariants (see [Ref04, p.36]). Static invariants are declared with the modifier *static* and may only refer to static fields whereas instance invariants do not have this restriction. Static Invariants must hold in every "visible" state i.e. before and after the execution of a (static or nonstatic) method or a constructor. The static Invariant of a type must be established by its static initialization.

Instance invariants of a type \( T \) are declared with the modifier *instance* and must hold in each visible state for each object of \( T \), i.e. before and after the execution of a method or constructor. A constructor must establish the instance invariants of the object it creates, and cannot assume them in its prestate (because the object does not yet exist at this point).

Invariants are by default instance invariants, if they are declared within a class, or static invariants, if they are declared within an interface.

JML offers the possibility of declaring *helper* methods and constructors that do neither have to preserve invariants nor can assume them to hold in their prestate. These *helper* methods have to be private.

Using *invariant_redundantly* indicates that an invariant is redundant but has no effect on the translation of this invariant as it will be completely ignored.

A method is only obliged to preserve invariants if it is called in a state that implies at least one of its preconditions, but if this is the case the invariant must be preserved even if the termination is abrupt\(^\text{10}\).

**Access Modifiers**

Invariant declarations can contain access modifiers. This does only affect which fields and methods can be used (namely those whose visibility is "\( \geq \)" the visibility defined by the given access modifier) in the given specification but has absolutely no effect on which methods have to preserve the invariants or not. Access modifiers are completely ignored by the implementation or in other words: All specifications are considered to be declared with the modifier *private*, so there is no restriction concerning the set of usable methods/fields.

As mentioned in section 2.1 it is assumed that the given JML specifications are syntactically valid. Since the official JML parser checks these visibility aspects, it is legitimate to perform no further checks on this issue.

\(^{10}\)by throwing an exception
3.3. MODEL METHODS

Inheritance

According to [Ref04] only instance invariants are inherited to subtypes. The translation differs in this issue from the JML-semantics as I can see no good reason for having no inheritance for static invariants. Consequently all kind of invariants are inherited in our semantics.

3.2.2 History Constraints

Syntax:

\[
\text{<history-constraint>} ::= \text{<constraint-keyword>} \text{<predicate>}; \\
\text{<constraint-keyword>} ::= \text{constraint} | \text{constraint_redundantly}
\]

Constraints are a means to describe how certain values are changed over time, which is achieved by the usage of \( \text{old} \). From a more pragmatic point of view one can regard constraints as additional postcondition that apply to every (non-helper) method declared in a type. Like with invariants we distinguish between static constraints, which have to be respected by static and instance methods and constructors, and instance constraints, which have to be respected by instance methods.

Once again declaring a method with the JML-modifier helper releases it of respecting constraints.

JML demands that the relation described by a constraint must be reflexive and transitive. This is not checked by KeY\textsuperscript{11} right now.

The handling of inheritance and access modifiers is the same as for invariants:

- In contrast to the JML-semantics, which defines inheritance only for instance constraints, all kind of constraints are inherited.
- Access modifiers are ignored

3.3 Model Methods

Model methods (see [Ref04, p.32]) are like model fields, which will be discussed later, a means to abstract from the concrete implementation. They are declared within JML annotations and are not visible to the implementation.

Unlike JML which allows describing the semantics by a methodbody or a specification, the translation to JavaDL only supports the specification variant right now.

Apart from its invisibility to the implementation a model method "behaves" like a normal method regarding inheritance, overriding and visibility (to the specification) aspects.

As mentioned in section 3.1.5 model methods have to be pure as they can only occur in specification clauses. This differs from the official JML semantics.

\textsuperscript{11}This would be done by providing a proof obligation for it
3.4 Model Fields

Model fields are declared with the JML modifier `model` and are used as a means to abstract from a concrete implementation. They are only visible to the specification and thus cannot occur in an implementation. As a consequence it is not possible to assign a value to a model field, because that would be an operation that is not free of side effects, which is not allowed in JML specifications. However the initial state of a model field can be described with the help of an initially clause.

One possible use of model fields is the specification of interfaces, where the absence of an implementation makes it necessary to have such abstract fields. Model fields used in interfaces (or classes) can then have representation that differ in the implementing classes (or subclasses).

3.4.1 The Represents Clause

Describing the relation between model fields is done by the represents clause.

There are basically two possibilities to express how a model field relates to an implementation:

- The model field is a representation for a real field or an expression of an adequate type. This is described by the `<-` operator.

  Example:

  ```java
  class ExampleForModel{
    int a;
    //@ public model int a_m;
    represents a_m <- (a>0 ? a : -a); @*
    ...
  }
  ```

- The represents clause describes a relation between the model field and an arbitrary number of real fields, which is done by the `\such_that` construct.

  Example:

  ```java
  class ExampleForModel{
    int a;
    //@ public model int a_m;
    represents a_m \such_that a<a_m && a_m<=2*a; @*/
    ...
  }
  ```

Note that the first kind of using the represents clause can be seen as a special case of the second variant, as `represents a <- b;` is always equivalent to `represents a \such_that a==b;`, so it is sufficient to consider the \such_that variant.

The possibilities to handle both kinds of represents clauses described in [BP03] form the basis of the following considerations for a correct treatment of model fields in JavaDL.
3.4. MODEL FIELDS

3.4.2 Translation of the Represents Clause

Of the two different approaches presented in [BP03], namely the axiomatic approach and the translation to a method, the axiomatic approach will be discussed first, as it better illustrates the problems, that can occur due to the translation of model fields to non-rigid variables.

Axiomatic Approach - a First Attempt

Given for example a represents clause of the form:

```java
//@ represents a_m <- a;
```

the naive approach would be to add an axiom\(^{12}\) of the form \(a_m = a\) to the pre- and postconditions that are the result of the translation of the requires, ensures or invariant clauses etc., but this would lead to incorrectness even in this simple case, as the following example shows us.

```java
class ExampleForModel{
    int a;
    //@ public model int a_m;   
    represents a_m <- a; @*/

    //@
    public normal_behavior
    requires true;
    ensures false;
    @*/
    public void test(){
        a=a+1;
    }
}
```

Figure 3.1: Example for an incorrect specification

It is obvious that the specification for \(m()\) shown in figure 3.1 cannot be satisfied. Nevertheless its (simplified) translation

\[
\vdash (a_m = a \land \text{true}) \rightarrow (a = a + 1); (a_m = a \rightarrow \text{false})
\]

is true as these derivation steps make clear:

\[
\vdash a_m = a \land a_m = a + 1 \vdash
\]

\[
\vdash a_m = a \rightarrow (a_m = a + 1 \rightarrow \text{false})
\]

\[
\vdash (a_m = a \land \text{true}) \rightarrow (a = a + 1); (a_m = a \rightarrow \text{false})
\]

If we have a closer look on what we have done, this incorrectness isn’t that surprising any more. The reason why this approach does not work is basically that for an axiom \(A(x)\) with occurrences of non rigid variables the formula

\(^{12}\)by adding an axiom \(A\) to the pre- and postcondition I mean transforming a formula of the form \(\phi \rightarrow (p)\psi\) into \((A \land \phi) \rightarrow (p)(A \rightarrow \psi)\)
24

CHAPTER 3. JML SPECIFICATIONS AND THEIR TRANSLATION TO JAVADL

A(x) → (p)A(x) is usually not true, as the nonrigid variables in A(x) might be changed by p, with the consequence that \{x|A(x)\} ≠ \{x|(p)A(x)\}, or in other words: x might have a different value in the poststate than in the prestate. For a correct axiomatic handling of represents clauses we have to use different variables x (or at least different values for x, which is possible as x is nonrigid) for the pre- and poststate.

In the case of represents clauses with <- translating a clause of the form 

represents a <- b; to an update \( f_a b g \) that is applied to the pre- and postcondition, or just substituting a with b, solves this problem, but this realization isn’t of any use in the case of such that.

Remark 3.4.1 Of course the more general solution that will be presented for such that-represents clauses in the next section can also be used for represents clauses using the <- notation, but still using updates as described above is a nice optimization, that leads to smaller and more readable formulas.

A Correct Axiomatic Approach

Let \( x \) be a model field and A(x) the axiom that holds for x. This is specified in a represents clause of the form:

// @ represents x \such_that A(x);

If we have a term \( \phi \) with occurences of \( x \), we have to replace it with:

\[
\pi_x(\phi)
\]

with

\[
\pi_x(\phi) = \begin{cases} 
\exists x' A(x)\{x\mapsto x'\} \land \forall x''(A(x) \rightarrow \pi_x'(\phi))\{x\mapsto x''\} & \text{if } x \text{ occurs in } \phi \\
\phi & \text{otherwise}
\end{cases}
\]

As we have to apply \( \pi_x \) also to subformulas of \( \phi \) which refer to different program states, i.e. subformulas that occur behind a modality the transformation \( \pi_x' \) is defined in the following way:

\[
\pi_x'(\phi) = \begin{cases} 
\phi & \text{if } \phi \text{ is a literal} \\
\pi_x'(\phi_1) \circ \pi_x'(\phi_2) & \text{if } \phi = \phi_1 \circ \phi_2, \text{ with } \circ \in \{\land, \lor\} \\
\forall y : t(\pi_x'(\phi_1)) & \text{if } \phi = \forall y : t(\phi_1) \text{ with } Q \in \{\forall, \exists\} \\
\forall x : t(\pi_x'(\phi_1)) & \text{if } \phi = \forall x : t(\phi_1) \\
\exists x : t(\pi_x'(\phi_1)) & \text{if } \phi = \exists x : t(\phi_1) \\
\forall y : t(\pi_x'(\phi_1)) & \text{if } \phi = \forall y : t(\phi_1) \\
\exists x : t(\pi_x'(\phi_1)) & \text{if } \phi = \exists x : t(\phi_1) \\
(\alpha) \pi_x(\psi) & \text{if } \phi = (\alpha) \psi \\
[\alpha] \pi_x(\psi) & \text{if } \phi = [\alpha] \psi
\end{cases}
\]

Here is a short motivation for \( \pi_x \):

- \( \exists x' A(x)\{x\mapsto x'\} \): We have to ensure that there is at least one \( x \), that satisfies the formula \( A(x) \). If this was not the case, \( \forall x''(A(x) \rightarrow \pi_x'(\phi))\{x\mapsto x''\} \) would be true irrespective of \( \phi \) and we could prove anything.

- \( \forall x''(A(x) \rightarrow \pi_x'(\phi))\{x\mapsto x''\} \): The part of this formula that might need some explanation is the newly introduced logic variable \( x'' \). The reason for this can be seen in Fig. 3.1. If the same (program)variable \( x \) occurs in the pre- and postcondition in an axiom \( A(x) \) then symbolically executing the program may lead to two contradictory terms for \( x \) (as seen in Fig. 3.1). By binding \( x \) to the logic variable \( x'' \) this is avoided.
3.4. MODEL FIELDS

Model Fields as Model Methods

As one can imagine the axiomatic approach potentially enlarges the resulting formula significantly if it contains several model fields being specified by such that clauses. One solution for this is to interpret model fields as (pure) methods, that do nothing else than returning the value (resp. reference) of the model field. We do not have to provide an implementation for this method, as we can use its specification as a lemma rule as described in [Ka03].

Let \( x \) be a model field, \( A(x) \) and the axiom that holds for \( x \). The specification of the model method \( x() \) that corresponds to \( x \) looks as follows:

- **precondition:** \( \exists x' A(x) \{ x \leftarrow x' \} \)
  
  This precondition is no surprise if we consider section 3.4.2. Like for the axiomatic approach we have to make sure that at least one \( x' \) exists, that satisfies the formula \( A(x) \).

- **postcondition:** \( A(x) \{ x \leftarrow \text{result} \} \)
  
  The postcondition is also relatively straightforward and probably needs no further explanation. The variable \( \text{result} \) denotes the result returned by \( x() \).

**Conclusion**

As we have seen, the axiomatic approach is definitely the way to go for \(<-\) clauses, as it can be easily realized by substituting or updating the model variable with its concrete value. For such that-clauses using axioms is no real alternative as this potentially leads to much bigger and unreadable formulas.

3.4.3 Inheritance of Model Fields

Model fields and their representations are inherited by subtypes, where they can also be overridden.

**Static vs. Dynamic Binding of Represents Clauses**

A question that arises in this context is what to do with occurances of model fields in inherited specifications\(^\text{13}\), whose representations in the subtype differs from its representation in the type the inherited specification is declared in:

- We can use the representations (as far as they exist), which are declared in the type the inherited methodspec is declared in (bind the representation statically), but then we risk that an old representation doesn’t make sense in the new context anymore.

- Or we use the representations, which are valid in the type, in which the method that inherits the specification, is located (dynamic binding). This might change the semantics of the specification.

**Remark 3.4.2 (Overriding of represents clauses)** A distinction between static and dynamic binding of representations is of course only relevant in cases

\(^{13}\)This consideration is not restricted to methodspecs. With invariants and constraints the same problem exists.
in which we have an overridden represents clause, otherwise there is no difference between the two alternatives. But after all an overridden represents clause is probably something that rarely happens since model fields are mostly declared in Interfaces\(^\text{14}\) where they usually have no initial represents clause (and thus it can not be overridden). A represents clause is then provided by the implementing classes.

What is used in our semantics is dynamic binding, which usually makes quite sense as the following scenario illustrates:

**Example 3.4.3 (Dynamic binding of representations)** Imagine we have a class `Account` which provides rudimentary banking functions and a model field `availableMoney`, which stands for the amount of money available:

```java
class Account{
    protected int balance;

    /** public model int availableMoney;
        @ public represents availableMoney <- balance;
        @*/

    ...

    /** ensures \result == availableMoney;
        @*/
    public int available(){
        return balance;
    }
}
```

The class `AdvancedAccount` extends `Account` and allows overdrawning the account:

```java
class AdvancedAccount{
    protected int overdraftLimit;

    /** public represents availableMoney <- balance+overdraftLimit;
        @*/

    ...

    public int available(){
        return balance + overdraftLimit;
    }
}
```

The method `available()` is overridden and inherits the specification from `Account`. As one can clearly see using static binding for representations would not make any sense here because the former represents clause just doesn’t fit to the new circumstances.

\(^{14}\)Where they are quite crucial because of the absence of “real” fields
One has to keep in mind that overriding a represents clause usually changes the semantics of the specifications the corresponding model field occurs in. As the following, admittedly pathological example shows:

Example 3.4.4 If we "bind" the represents clause dynamically the method div in SubTest doesn’t meet the specification anymore, although it is identical to the method div in Test.

class Test{
    int a;

    /** model int m_a;
     * represents m_a <- a;
     */

    /** requires m_a != 0;
        ensures true;
        */
    public int div(int b){
        return b / a;
    }
}

class SubTest extends Test{

    /** represents m_a <- a - 1;
        */

    public int div(int b){
        return b / a;
    }
}

3.5 Types

JML uses the same primitive types as Java, built-in types of Java and class types known in the context of a java program. So the translation of types is canonical and JML expressions are always translated to JavaDL terms of the corresponding type, except for expressions of type boolean, which can be translated to formulas.
Chapter 4

The Translation of JML Expressions

The following section will give an overview over the semantics of expressions in JML and how they are translated to JavaDL. The syntax given for the expressions in this section reflects the syntax the translation can handle. So it is possible that it is in some cases just a subset of the real JML-syntax.

The translation of an expression $expr$ will be denoted with $\langle expr \rangle$.

$$\langle pred-or-not \rangle ::= \langle predicate \rangle \mid \text{not.specified}$$

$$\langle predicate \rangle ::= \langle spec-expression \rangle$$

$$\langle spec-expression \rangle ::= \langle expression \rangle$$

$$\langle expression \rangle ::= \langle conditional-expr \rangle$$

An expressions which is the body of a specification clause has to be of type boolean. These expressions are always directly translated to formulas. As an optimization one could imagine, to translate those boolean expressions, which occur as an argument of a method, to a term of sort boolean. On the other hand this is not always possible, as such an expressions may contain special JML constructs like quantifiers, which must be translated to a formula. More on this and the translation of method calls within specifications can be found in section 4.14.8.

4.1 Conditional Expressions

Syntax

A conditional expression is an expression of the form:

$$\langle conditional-expr \rangle ::= \langle equivalence-expr \rangle$$

$$[? \langle conditional-expr \rangle : \langle conditional-expr \rangle]$$

Semantics

The semantics of this expression is the same as in java, i.e. the expression $P \ ? \ Q \ : \ R$ evaluates to $Q$ if $P$ is true and to $R$ otherwise.
4.2. EQUIVALENCE EXPRESSIONS

Translation

Rule 4.1.1 (Translation of Conditional Expressions) A conditional expression P ? Q : R is translated as follows:

\[ [P \, ? \, Q : R] := \begin{cases} \text{typeof}(Q) = \text{boolean} \\ \text{typeof}(Q) \neq \text{boolean} \end{cases} \]

where \( m() \) is a newly introduced pure model method with a unique name.

If the second case occurs and a model method \( m() \) is introduced, the following normal-behavior-specification for this method is created:

\[
\text{normal_behavior} \\
\quad \text{ensures} \ (P \implies \\result = Q) \&\& \ (!P \implies \\result = R); 
\]

4.2 Equivalence Expressions

Syntax

JML has the built-in operators \( <=> \) and \( <=!=> \) to express equivalence and antivalence.

\[
\text{<equivalence-expr>} ::= \text{<implies-expr>} \\
\quad ( ( \text{<==>} | \text{<=!==>} ) \text{<implies-expr>} )^* 
\]

Semantics

- \( P<==>Q \) is true if and only if \( P \) and \( Q \) both evaluate.
- \( P<=!=>Q \) is true if and only if \( P<==>Q \) is false.

Translation

Rule 4.2.1 (Translation of Equivalence Expressions) The translation of equivalence expressions is, as one might already suspect, quite canonical:

\[
[P \, <==> \, Q] := \text{[}[P] \leftrightarrow [Q]\text{]} \\
[P \, <=!=> \, Q] := \neg([P] \leftrightarrow [Q]) 
\]

4.3 Implies Expressions

Syntax

\[
\text{<implies-expr>} ::= \text{<logical-or-expr>} \\
\quad [ \implies \text{<implies-non-backward-expr>} ] \\
\quad | \text{<logical-or-expr>} (<!= \text{<logical-or-expr>})^* 
\]

Semantics

The JML implication has the same semantics as the implication in first-order logic. \( P<==>Q \) means the same as \( Q=>=P \).

\(^1\) As expressions in JML specifications must not have side effects there is no such thing like an evaluation order we have to care about.
Translation

Rule 4.3.1 (Translation of Implies Expressions)

\[ [\text{P} \implies \text{Q}] := [\text{P}] \rightarrow [\text{Q}] \]
\[ [\text{P} \iff \text{Q}_1 \iff \cdots \iff \text{Q}_n] := [\text{Q}_n] \rightarrow \ldots [\text{Q}_1] \rightarrow [\text{P}] \]

4.4 Logical Or Expressions

Syntax

\(<\text{logical-or-expr}> ::=: <\text{logical-and-expr}> \]
\[ [\text{P} \lor \text{Q}] := [\text{P}] \lor [\text{Q}] \]

Semantics
The semantics of the logical or in JML is exactly the same as the semantics of its java counterpart, i.e. \(\text{P} \lor \text{Q}\) where \(\text{P}\) and \(\text{Q}\) are expressions of type boolean is true if and only if \(\text{P}\) is true or \(\text{Q}\) is true.

Translation

Rule 4.4.1 (Translation of Logical Or Expressions)

\[ [\text{P} \lor \text{Q}] := [\text{P}] \lor [\text{Q}] \]

4.5 Logical And Expressions

Syntax

\(<\text{logical-and-expr}> ::= <\text{inclusive-or-expr}> \]
\[ [\text{P} \land \text{Q}] := [\text{P}] \land [\text{Q}] \]

Semantics
The semantics of \(\land\) in JML is the same as in Java, i.e. \(\text{P} \land \text{Q}\) evaluates to true if and only if \(\text{P}\) is true and \(\text{Q}\) is true.

Translation

Rule 4.5.1 (Translation of Logical Or Expressions)

\[ [\text{P} \land \text{Q}] := [\text{P}] \land [\text{Q}] \]

4.6 Inclusive Or Expressions

Syntax

\(<\text{inclusive-or-expr}> ::= <\text{exclusive-or-expr}> \]
\[ [\text{P} \lor \text{Q}] := [\text{P}] \lor [\text{Q}] \]

Semantics
The semantics of \(\lor\) in JML is the same as in Java, i.e. \(\text{P} \lor \text{Q}\) evaluates to true if and only if \(\text{P}\) is true or \(\text{Q}\) is true.
4.7. EXCLUSIVE OR EXPRESSIONS

Semantics

The inclusive or denotes a bitwise or when applied on variables of a primitive type that is not boolean.

When applied on boolean variables its semantics is basically that of a logical or but with strict evaluation. However as expressions in JML must be side effect free, that’s nothing we have to care about and we can treat it just like a logical or in this case.

Translation

Rule 4.6.1 (Translation of Inclusive Or Expressions)

\[ [P \lor Q] := \begin{cases} ([P] \lor [Q]) & \text{typeof}(P) = \text{boolean} \\ f_{\text{xorlong}}([P], [Q]) & \text{typeof}(P) = \text{long} \text{ or } \text{typeof}(Q) = \text{long} \\ f_{\text{xorint}}([P], [Q]) & \text{otherwise} \end{cases} \]

where \( f_{\text{xorint}} \) and \( f_{\text{xorlong}} \) are the appropriate functions for a bitwise or in the selected integer semantics.

Note: This and any other translation that distinguishes between operations for long and int can of course only work if we are running KeY in the Java Semantics.

4.7 Exclusive Or Expressions

Syntax

\[ <\text{exclusive-or-expr}> ::= <\text{and-expr}> [ `\text{^}' <\text{exclusive-or-expr}> ] \]

Semantics

Like in the case of the inclusive or the exclusive or can be a logical or a bitwise operation depending on its operands. If \( P \) and \( Q \) are boolean, \( P^\lor Q \) evaluates to true if and only if \( P \) is true and \( Q \) is false or \( P \) is false and \( Q \) is true.

If \( P \) and \( Q \) have a non-boolean primitive type \( ^\lor \) is a bitwise XOR or in other words an addition in \( \mathcal{GF}(2)^n \) with \( n = 64 \) if one of the operands has type long or \( n = 32 \) otherwise.

Translation

Rule 4.7.1 (Translation of Exclusive Or Expressions) Again we have to distinguish between the translation to a formula and to a term:

\[ [P \land Q] := \begin{cases} ([P] \land \neg [Q]) \lor (\neg [P] \land [Q]) & \text{typeof}(P) = \text{boolean} \\ f_{\text{xorlong}}([P], [Q]) & \text{typeof}(P) = \text{long} \text{ or } \text{typeof}(Q) = \text{long} \\ f_{\text{xorint}}([P], [Q]) & \text{otherwise} \end{cases} \]

where \( f_{\text{xorint}} \) and \( f_{\text{xorlong}} \) are the appropriate functions for a bitwise xor in the selected integer semantics.
4.8 And Expressions

Syntax

<and-expr> ::= <equality-expr> [ & <and-expr> ]

Semantics

Analogous to ∧ and || the and-operator & can denote a bitwise or a logical operator.

Translation

Rule 4.8.1 (Translation of And Expressions)

\[ [P \& Q] := \begin{cases} [P] \land [Q] & \text{typeof}(P) = \text{boolean} \\ f_{\text{andlong}}([P], [Q]) & \text{typeof}(P) = \text{long} \\ f_{\text{andint}}([P], [Q]) & \text{otherwise} \end{cases} \]

where \( f_{\text{andint}} \) and \( f_{\text{andlong}} \) are the appropriate functions for a bitwise and in the selected integer semantics.

4.9 Equality Expressions

Syntax

<equality-expr> ::= <relational-expr> [ == <equality-expr> ]
| <relational-expr> [ != <equality-expr> ]

Semantics

The semantics is the same as in Java.

Translation

As a relational-expr may be boolean, a case distinction between a translation to an equality-operator and to an equivalence-operator is necessary. Note that \( P==Q \) ( \( P!=Q \) resp.) is equivalent to writing \( P<==>Q \), \( P<=!=>Q \) (resp.) if \( P \) and \( Q \) are boolean expressions.

Rule 4.9.1 (Translation of Equality Expressions)

\[ [P == Q] := \begin{cases} [P] \leftrightarrow [Q] & \text{typeof}(P) = \text{boolean} \\ [P] = [Q] & \text{typeof}(P) \neq \text{boolean} \\ [P != Q] := \neg [P == Q] \]
4.10 Relational Expressions

Syntax

<relational-expr> ::= <shift-expr> < <shift-expr>
| <shift-expr> > <shift-expr>
| <shift-expr> <= <shift-expr>
| <shift-expr> >= <shift-expr>
| <shift-expr> [ <instanceof type-spec> ]

Semantics

Same as in Java.

Translation

Terms with <, >, <= or >= as top level symbol are translated canonically to predicates. Terms of the form \texttt{t instanceof type} are translated the following way:

\[ \texttt{[t instanceof type]} := instance_{type}([t]) \]

where \texttt{instance}_{type} is the corresponding predicate in JavaDL for \texttt{instanceof type}.

4.11 Shift, additive, multiplicative and unary Expressions

Syntax

<shift-expr> ::= <additive-expr> ( <shift-op> <additive-expr> )*
<shift-op> ::= << | >> | >>>
<additive-expr> ::= <mult-expr> ( <additive-op> <mult-expr> )*
<additive-op> ::= + | -
<mult-expr> ::= <unary-expr> ( <mult-op> <unary-expr> )*
<mult-op> ::= * | / | %
<unary-expr> ::= ( <type-spec> ) <unary-expr>
| + <unary-expr>
| - <unary-expr>
| <unary-expr-not-plus-minus>

Semantics

Same as in Java.

Translation

The operators are translated to the corresponding depending on the active integer semantics. Which means again (as in sections 4.6, 4.7 or 4.8) for the Java Semantics that we have to distinguish between 64 bit (\texttt{long}) and 32 bit (\texttt{int}) operations. 64 bit operations are used, if and only if at least one of the operands is of type \texttt{long}, otherwise 32 bit operations are used.
4.12 Quantifiers

Besides the actual quantifiers $\forall$ and $\exists$ JML offers so called generalized quantifiers which are nothing else then the functions $\min$, $\max$, $\sum$ and $\prod$ defined over a set of arguments.

Syntax

$$
<\text{spec-quantified-expr}> ::= ( <\text{quantifier}> <\text{quantified-var-decls}> ; [ [ <\text{predicate}> ] ; ] <\text{spec-expression}> )
$$

$$
<\text{quantifier}> ::= \forall | \exists | \max | \min | \num_of | \product | \sum
$$

Semantics

"Real" Quantifiers A quantified expression $(\forall <\text{type}> ~x; R(~x); P(~x))$, where $~x$ denotes an arbitrary number of variables of the same type and $R(~x)$ and $P(~x)$ are boolean expressions, has the semantics:

$(\forall <\text{type}> ~x; R(~x); P(~x))$ is true if and only if for all tuples $~x$, that satisfy the range predicate $R(~x)$, $P(~x)$ is true.

For existentially quantified expressions we analogously get the semantics:

$(\exists <\text{type}> ~x; R(~x); P(~x))$ is true if and only if a tuple $~x$ exists for with $R(~x)$ and $P(~x)$ are true. If $R(~x)$ is omitted it is true by default.

When quantifying over variables that have an object- or array-type, the range of this quantifier is implicitly restricted to arrays/objects that are already created and the constant $null$. In JavaDL this has to be expressed explicitly. Therefore we have to use the quantifiers $\_\forall$ and $\_\exists$, where $\_\exists A$ (resp. $\_\forall A$) is syntactic sugar for $\exists x(x = null \lor x \neq null \land x.<\text{created}> = true) \land A$ (resp. $\forall x(x = null \lor x \neq null \land x.<\text{created}> = true) \lor A$).

Generalized Quantifiers In the following cases $P (~x)$ has a numerical type$^2$.

$(\min <\text{type}> ~x; R(~x); P(~x))$ has the meaning $\min\{P(~x) \mid R(~x)\}$. If the set $\{P(~x) \mid R(~x)\}$ is empty the largest number with the type of $P(~x)$ is taken as result (i.e. either $\text{Integer.MAX_VALUE}$ or $\text{Long.MAX_VALUE}$ depending on $P(~x)$).

$(\max <\text{type}> ~x; R(~x); P(~x))$ is then $\max\{P(~x) \mid R(~x)\}$ if the set $\{P(~x) \mid R(~x)\}$ isn’t empty and the smallest value with the type of $P(~x)$ otherwise.

The semantics of $\sum$ and $\prod$, which are currently not supported by the translation, is as follows:

- $(\sum <\text{type}> ~x; R(~x); P(~x)) = \sum_{x \in R(~x)} P(~x)$

- $(\prod <\text{type}> ~x; R(~x); P(~x)) = \prod_{x \in R(~x)} P(~x)$

$^2$as there is no support for floating point data types in KeY we can only handle cases where $P(~x)$ has the sort $\text{int}$.
Translation

4.12 Rule 4.12.1 (Translation of Quantifiers) It is clear that "real quantifiers" and generalized quantifiers are translated in a fundamentally different way. The translation of the existential and universal quantifier is quite straightforward:

- For all: $\forall x : R(x); P(x)) := \forall x (R(x) \rightarrow P(x))$
- Exists: $\exists x : R(x); P(x)) := \exists x (R(x) \land P(x))$

Where $\exists x$ and $\forall x$ stands for $\exists x_1 \ldots \exists x_n$ or $\forall x_1 \ldots \forall x_n$ respectively, with $x = (x_1, \ldots, x_n) = ([x_1]_1, \ldots, [x_n]_n) = [x]_n$.

For min and max we have to note the following consideration: Let $y_{\text{min}}$ be a value with $y_{\text{min}} = (\forall x : R(x); P(x))$ then the formula $\psi(y_{\text{min}})$ must hold:

$$\psi_{\text{min}}(y_{\text{min}}) := \neg(\exists x : R(x)) \land y_{\text{min}} = \max y_{\text{max}}(P(x)) \land y_{\text{min}} \leq P(x) \land \exists x : (R(x) \land P(x) = y_{\text{min}})$$

with $\max y_{\text{max}}(P(x))$ being the largest value a number of the type of $P(x)$ can have.

For an $y_{\text{max}}$ with $y_{\text{max}} = (\exists x : R(x); P(x))$ we analogously have:

$$\psi_{\text{max}}(y_{\text{max}}) := \neg(\exists x : R(x)) \land y_{\text{max}} = \min y_{\text{min}}(P(x)) \land y_{\text{max}} \geq P(x) \land \exists x : (R(x) \land P(x) = y_{\text{max}})$$

Now we have basically two alternatives:

1. We can translate a max-expression to a new program variable $y_{\text{max}}$ and add in an appropriate way the term $\psi_{\text{max}}(y_{\text{max}})$ as an axiom to the formulas $y_{\text{max}}$ occurs in.

2. The max-expression is translated to a pure model method $y_{\text{max}}()$ that has a normal_behavior specification with $\psi_{\text{max}}$ result as postcondition and of course the type of $P(x)$ as return type. result denotes the value returned by $y_{\text{max}}()$.

The second alternative is the one that is implemented, as it turned out to be the more practicable way to do it, regarding among other things the readability of the translation result.

The translation of $\langle \min \rangle$ is done in the same way.

4.13 The Construct $\backslash$nonnull_elements

$\backslash$nonnull_elements is used to express that an array reference and the elements of the array it points to are not null (see [LBR03, p.56]). JML allows only expressions of non-primitive array types as arguments of $\backslash$nonnull_elements.

For a 1-dimensional array myArray, $\backslash$nonnull_elements(myArray) is equivalent to:

- conjunctively if $y_{\text{max}}$ occurs in a precondition, $\psi_{\text{max}}(y_{\text{max}}) \rightarrow \text{post}$ if $y_{\text{max}}$ occurs in the postcondition \text{post}, . . .
myArray != null &&
(\forall int i; 0 <= i && i < myArray.length;
  myArray[i] != null)

This leads to the translation:

**Rule 4.13.1 (Translation of \nonnullelements)** Let o be an object- or array-reference. Then \nonnullelements(o) can be recursively defined in the following way:

\[
\text{\nonnullelements}(o) := \left\{ \begin{array}{ll}
\text{o}! = \text{null} \&\& (\forall \text{int } i; 0 \leq i \&\& i < \text{o}.length; \text{\nonnullelements(o[i])}), & \text{if typeof(o) is subtype of Object[]} \\
\text{o}! = \text{null}, & \text{otherwise}
\end{array} \right.
\]

where \text{\nonnullelements}(o) stands for \text{\nonnullelements} and typeof(o) denotes the static type of o.

**Remark 4.13.2 (\nonnullelements and java.lang.Object)** In Java Array types are subtypes of java.lang.Object. Thus if the variable o had the (static) type Object[] and typeof(x) denoted the runtime type of x, it would not be possible to create the formula shown in rule 4.13.1 in such a "static" way, as the depth of the recursion would depend on the runtime type of o.

One possible solution in this case could be defining a query \text{\nonnullelements} which has the specification:

```java
/*@ public normal_behavior
@ ensures o != null &&
@ ((o instanceof Object[]) ==>
@ (\forall int i; 0 <= i && i < (Object[] o).length;
@  \text{\nonnullelements}((Object[] o)[i])));
@ public pure model boolean \text{\nonnullelements}(Object o);
@*/
```

It is unclear whether JML refers to the static or to the runtime type in the context of \text{\nonnullelements}. For KeYJML it is assumed that the static type is meant\(^4\), which has the effect that in this case the translation presented in rule 4.13.1 is the correct one.

### 4.14 Atomic Expressions

Since boolean expressions are always translated to formulas, we have to distinguish between boolean and non-boolean atomic expressions.

That is why the translation of an atomic expression \text{expr} of type boolean always ends up in a formula of the form \text{expr_{JavaDL} = TRUE}.

For the sake of readability we will use the following notation for this purpose:

\[
B(\text{expr}) := \begin{cases} 
\text{expr}, & \text{sortof(\text{expr}) \neq boolean} \\
\text{expr = TRUE}, & \text{sortof(\text{expr}) = boolean}
\end{cases}
\]

\(^4\)One evidence that supports this assumption might be that the official JML parser does not allow variables of type Object as parameters of \text{\nonnullelements}, which wouldn’t make sense if JML referred to the runtime type in the context of \text{\nonnullelements}
4.14. ATOMIC EXPRESSIONS

4.14.1 \texttt{old}

The JML expression \texttt{old(expr)} may appear in ensures and signals clauses and history constraints. It denotes the evaluation of the expression \texttt{expr} in the state before the execution of the specified method. In this respect it is comparable to OCL’s \texttt{@pre} operator, whose translation to JavaDL is discussed in [BBS01] and [Ka03]. Contrary to \texttt{@pre} which refers only to the top level function symbol, \texttt{old} refers to the whole expression. Thus for example the JML expression \texttt{old(this.o.attr)} is equivalent to \texttt{this.o@pre.attr@pre}. Furthermore \texttt{old} can be applied to arbitrary expressions, not only to locations and queries as in the case of \texttt{@pre}.

Rule 4.14.1 (The Translation of \texttt{old}) As a first step \texttt{old(expr)} is translated to a newly introduced program variable \texttt{old_expr} if \texttt{expr} contains no free variables, or a function term \texttt{old_expr(x_1, \ldots, x_n)}, where \texttt{old_expr} is a new function symbol, if \texttt{expr} contains the free variables \texttt{x_1; \ldots; x_m}:

\[
|\texttt{old(expr)}| = \begin{cases} 
\mathcal{B}(\texttt{old_expr}), & \text{if expr contains no free variables} \\
\mathcal{B}(\texttt{old_expr}(x_1, \ldots, x_n)), & \text{if expr contains free variables } x_1, \ldots, x_n
\end{cases}
\]

Afterwards the transformation \(\tau_{\text{old}}\) is applied to the resulting formula.

Definition 4.14.2 (The Transformation \(\tau_{\text{old}}\)) Let \(\pi = \phi \rightarrow (p)\psi\) be a formula resulting from the translation of a JML specification with occurrences of expressions \texttt{old(expr_1), \ldots, old(expr_n)}. Then

\[
\tau_{\text{old}}(\pi) := \forall y_1, \ldots, y_m (\phi \land \bigwedge_{i=1}^{m} \Omega(y_i, \texttt{old_expr}_i) \land \bigwedge_{j=m+1}^{n} \Omega_{\phi}(\texttt{old_expr}_j) \rightarrow (p)\psi)
\]

where

- \texttt{old_expr_i} are variables resulting from the translation of the expressions \texttt{old(expr_i)}.
- \texttt{expr_1, \ldots, expr_m} contain no free variables.
- \texttt{expr_{m+1}, \ldots, expr_n} contain free variables.
- if \texttt{expr} contains no free variables and is not \texttt{boolean} then
  \[\Omega(y, \texttt{old_expr}) := y \doteq \texttt{expr} \land y \doteq \texttt{old_expr}\]
- if \texttt{expr} contains no free variables and is \texttt{boolean} then
  \[\Omega(y, \texttt{old_expr}) := y \doteq \text{TRUE} \leftrightarrow \texttt{expr} \land y \doteq \texttt{old_expr}\]
- if \texttt{expr} contains the free variables \texttt{x_1, \ldots, x_m} and is not \texttt{boolean} then
  \[\Omega_{\phi}(\texttt{old_expr}) := \forall x_1 \ldots x_m ([\texttt{expr}] \doteq \texttt{old_expr}(x_1 \ldots x_m))\]
- if \texttt{expr} contains the free variables \texttt{x_1, \ldots, x_m} and is \texttt{boolean} then
  \[\Omega_{\phi}(\texttt{old_expr}) := \forall x_1 \ldots x_m ([\texttt{expr}] \leftrightarrow (\text{TRUE} \doteq \texttt{old_expr}(x_1 \ldots x_m)))\]
Remark 4.14.3 As an optimization it is possible to use updates instead of equations which only works for the cases in which we have a program variable. The resulting transformation \(\tau_{update}\) is:

\[
\tau_{update}(\pi) :=
(\forall y_{k+1}, \ldots, y_m (\phi \land \bigwedge_{j=k+1}^m y_j = \text{TRUE} \iff [\text{expr}_1] \land \bigwedge_{j=m+1}^n (\Omega_v \text{old}_{\text{expr}_j}) \rightarrow
(p)\psi)\]

where

- \(\text{expr}_1, \ldots, \text{expr}_k\) are nonboolean expressions that do not contain free variables.
- \(\text{expr}_{k+1}, \ldots, \text{expr}_m\) are boolean expressions that do not contain free variables.
- \(\text{expr}_{m+1}, \ldots, \text{expr}_n\) contain free variables.
- \(\Omega_v\) is defined as before.

The transformation \(\tau_{old}\) is just a simple adaption of the transformation \(\tau_v\) (at least for expressions without free variables) introduced in [BBS01] and is used to generate formulas that form the basis for the generation of lemma rules for the specified methods. For the generation of proof obligations \(\tau_{update}\) is used.

In order to understand that it makes semantically no difference which of the two transformations we choose, we have to make ourselves clear that \(\vdash p = a \rightarrow \langle \text{prog} \rangle \phi\) holds if and only if \(\vdash \{ p := a \} \langle \text{prog} \rangle \phi\) holds:

**Proof 4.14.4** Let \(p = x \rightarrow \langle \text{prog} \rangle \phi\) and \(\{ p := a \} \langle \text{prog} \rangle \phi\) be JavaDL formulas and \(\mathcal{K} = (\mathcal{S}, \rho)\) the underlying Kripke structure, where \(\mathcal{S}\) is a set of states, \(\rho\) is a partial function from programs to transition relations (in our case functions as Java is deterministic). If \(p\) is a program variable that occurs neither in \(\Gamma\) nor in \(\Delta\) then \(\Gamma \vdash p = a \rightarrow \langle \text{prog} \rangle \phi\), \(\Delta\) holds if and only if \(\Gamma \vdash \{ p := a \} \langle \text{prog} \rangle \phi\), \(\Delta\) holds:

\[
\Gamma \vdash p = a \rightarrow \langle \text{prog} \rangle \phi, \Delta
\]

\[
\iff
f.a. s \in \mathcal{S} : \text{if } s \vdash p = a \land \Gamma \text{ then } s \vdash \langle \text{prog} \rangle \phi, \Delta
\]

\[
p \text{ doesn't occur in } \Gamma
\]

\[
f.a. s' : \text{if } s' \vdash \Gamma \text{ then }
\]

\[
\quad (if \quad \exists s \text{ with } (s', s) \in \rho(p = a) \quad \text{ then } s' \vdash \{ p := a \} \langle \text{prog} \rangle \phi, \Delta)
\]

\[
always \text{ true}
\]

\[
p \text{ doesn't occur in } \Delta
\]

\[
\iff
f.a. s' : s' \vdash \Gamma \text{ then } s' \vdash \{ p := a \} \langle \text{prog} \rangle \phi, \Delta
\]

\[
\iff
\Gamma \vdash \{ p := a \} \langle \text{prog} \rangle \phi, \Delta
\]

\[\blacksquare\]

4.14.2 The Result Construct \result

\result\ is a JML-construct that can occur in method specifications and denotes the value returned by the method the specification belongs to.
Rule 4.14.5 (Translation of result)

\[
\result := \mathcal{B} (\result)
\]

Where result is a new program variable of the same type as result and with a unique name.

4.14.3 Parameters of Signals Clauses

Signals clauses can be declared with a parameter representing the thrown exception, that then can be used in the corresponding exceptional postcondition. For a given signals clause of the form signals (E e) P; the occurrences of e in P is what we are talking about in this paragraph.

Parameters of signals clauses are translated to program variables. The occurrences of such a parameter in the body of a signals clauses are then directly translated to this program variable.

Rule 4.14.6 (Translation of Parameters of Signals Clauses)

\[
[e] := e
\]

where e is a variable declared by a signals clause and e is a program variable in JavaDL of the same type.

4.14.4 Parameters

The parameters of a method can be used in the requires, ensures, signals and diverges clause but not in the assignable clause as the assignment to a parameter would not be visible to the caller anyway and consequently is not of any relevance for him. For the same reason parameters in specifications are always evaluated in the prestate which means writing p in a clause concerning the postcondition (ensures, signals), is equivalent to writing \old(p).

Rule 4.14.7 (Translation of parameters)

\[
[p] := \begin{cases} 
\mathcal{B}(p), & \text{in requires or diverges clauses} \\
\old(p), & \text{in ensures or signals clauses}
\end{cases}
\]

Where p, is the program variable representing p in the program context.

Remark 4.14.8 The transformations \tau_{old} and \tau_{update} do not distinguish between the explicit usage of \old and the implicit application of \old to parameters.

4.14.5 The Constant this
this is translated to a new program variable self.
4.14.6 Attribute and Array References

As mentioned in section 1.1.1, attributes and arrays are represented in JavaDL as non-rigid functions. Attribute and array references are canonically translated to JavaDL function terms. If the attribute reference refers to a nonstatic attribute of the enclosing type, the program variable self mentioned in the preceding paragraph is used as prefix.

If the reference is boolean the proceeding is the same as for other atomic expressions.

4.14.7 Method References

Method references are canonically translated to JavaDL queries. In this context it is not relevant if the methods concerned are implemented methods or "just" model methods. For methods with return type boolean the problem, as well as its solution, is the same as in the preceding paragraphs.

4.14.8 Boolean Expressions as Arguments of Method Calls

As mentioned before boolean expressions are always translated to formulas. This inevitably becomes a problem, if a boolean expression \texttt{expr} happens to be an argument of a method call, since a formula as argument of a function would of course neither be syntactically possible nor make any sense.

Therefore we translate \texttt{expr} to a term \texttt{m()}. Where \texttt{m} is a newly introduced query which meets the specification:

\begin{verbatim}
normal_behavior
  ensures \result == expr;
public model boolean m();
\end{verbatim}

Remark 4.14.9 If \texttt{expr} is just a variable or an attribute reference one could of course refrain from translating it to a formula and instead translate it directly to a JavaDL variable or attribute term\footnote{Which is actually done by the implementation in such simple cases.}, which surely makes more sense than the introduction of a new query.

Nevertheless there are cases when it is not possible to translate a JML expression of type boolean to a boolean JavaDL term. This is for instance the case for quantified expressions.
Chapter 5

Proof Obligations

This chapter will illustrate how proof obligations are generated from JML specifications. This will be done by starting with simple cases, containing just a pre-post-pair and then incrementally extending them. As far as method specifications are concerned, we are always considering desugared heavyweight specifications in this chapter.

The proof obligations presented in this chapter always cover only certain aspects of the correctness requirements a method has to fulfil. The invariant proof obligation for instance only expresses that a method has to preserve the invariants of the type it is contained in but not that its execution preserves the invariants of every other existing object or class.

5.1 Proving Method Specification Cases

Let $m$ be a nonstatic method with the signature $m\text{\texttt{(type}_1,\ldots,\text{\texttt{type}}_n)}$ declared in the type $\text{\texttt{ContextType}}$. Let the returntype of $m$ be $\text{\texttt{type}}_r$.

For nonstatic method the formula $\text{\texttt{pre}}_{\text{\texttt{impl}}}$ is added to the precondition, where $\text{\texttt{pre}}_{\text{\texttt{impl}}}$ stands for the implicit precondition:

\[
\text{\texttt{self}} \neq \text{\texttt{null}} \land \text{\texttt{self.<created>}} = \text{\texttt{TRUE}} \land \\
\text{\texttt{ContextType.<classInitialized>}} = \text{\texttt{TRUE}}
\]

In the case of a static method $\text{\texttt{pre}}_{\text{\texttt{impl}}}$ stands for:

\[
\text{\texttt{ContextType.<classInitialized>}} = \text{\texttt{TRUE}}
\]

Methods with returntype $\text{\texttt{void}}$ will not be discussed separately, since the only change in the created JavaDL formula would be the absence of the $\text{\texttt{result}}$ variable in the $\text{\texttt{MethodBodyStatement}}$. Another change is made for proof obligations for static methods and constructors where the quantification $\forall s : [\text{\texttt{ContextType}}]$ is superfluous. These and other canonical modifications of proof obligation terms will not be discussed in the following sections, since they are rather obvious and make no fundamental difference regarding the terms concerned.
5.1.1 Normal Behavior Cases without Diverges

Assume there is a specification case for \( m \) of the form:

```java
/*@ public behavior
@ requires A;
@ ensures B;
@ diverges false;
@ signals (Exception) false;
@*/
```

which is the desugared version (see also [Ref04, p.50] or section 3.1) of the normal behavior specification case:

```java
/*@ public normal_behavior
@ requires A;
@ ensures B;
@*/
```

We now have to prove that if \( m \) is called in a state in which the precondition \( A \) and all invariants\(^2\) of any existing type or object, \( m \) terminates and the postcondition \( B \) holds (after the termination). This leads us to the JavaDL formula:

\[
\forall s :\left\langle ContextType \right\rangle \{ (self := s) (\\text{Q}_{rep}) \text{Q}_{param}^{\text{update}} \text{inv} \land \text{preimpl} \rightarrow (\text{ContextType}(\text{result}) :: self.\text{m}(p_1, ..., p_n);) \text{Q}_{rep}([B])) \}
\]

where

- \( \text{Q}_{rep}\phi \) is an abbreviation for \( \{ m_1 := m_1(), ..., m_n := m_n() \} \phi \) where \( \phi \) is a formula, \( m_1, ..., m_n \) are modellfields occurring in \( \phi \) and \( m_1(), ..., m_n() \) are the queries generated from the corresponding represents clauses (of \( \text{ContextType} \)).

- \( \text{Q}_{param}\phi \) is an abbreviation for \( \forall p_1 : s_1' \cdots \forall p_n : s_n' \{ p_1 := p_1', \ldots, p_n := p_n' \} \phi \), where \( \phi \) is again a formula, \( p_1, \ldots, p_n \) are program variables of the sorts \( s_1, \ldots, s_n \) representing the parameters of \( m \), and \( p_1', \ldots, p_n' \) are logic variables of the sorts \( s_1, \ldots, s_n \).

- \( \text{inv} \) is a formula representing the fact that all instance invariants for every existing object and all static invariants for every type, that has finished its static initialization, hold. Thus \( \text{inv} \) is:

\[
\text{inv} := \bigwedge_{T : T \in T} \forall o : T(\{ \text{self}_T := o \} \text{Q}_{rep}(T(\{(o \neq \text{null} \land o.\text{created} = \text{true} \rightarrow [\text{inv}_0]) \land (T, \text{classInitialized} = \text{TRUE} \rightarrow [\text{inv}_T])\}))
\]

---

\(^1\)We will ignore assignable clauses here as well as in the whole chapter, since they are not relevant for the kind of proof obligations we discuss here. For more on proving the assignable clause itself see [Sa04].

\(^2\)If \( m \) was a helper method we couldn’t assume any invariant to hold here. For details on this issue see section 3.2.1 or [Ref04, p.40]
5.1. PROVING METHOD SPECIFICATION CASES

- $T$ is the set of known classes and interfaces.
- $[\text{inv}_o]$ is a conjunction of the invariants applicable to $o$, i.e. the static and instance invariants of $T$.
- $[\text{inv}_r]$ are the static invariants of $T$.
- $Q_{rep_T}$ only differs from $Q_{rep}$ in that respect that the represents clauses it refers to are those of type $T$.
- $\text{self}_T$ is a program variable of type $T$ introduced during the translation of the specification of $T$ according to section 4.14.5.

- $\tau_{\text{update}}$ is the transformation described in section 4.14.1.
- $\text{self}$ is a program variable of type ContextType (see also 4.14.5).

**Remark 5.1.1 (Invariants and Preconditions)** A method specification case only refers to prestates in which besides the precondition given by the requires clause also all applicable invariants hold (see [Ref04, p.51]). Therefore conjoining inv with the precondition, as it was done here, is no arbitrary restriction, it’s mandatory.

However it is often sufficient for proving a method specification to assume just the invariants of the containing class. Doing this can be helpful in some cases, since the term inv can become rather big.

Thus it is possible to create proofobligations in which inv only consists of the applicable invariants of ContextType, but one should note that in general this is not complete.

### 5.1.2 Behavior Cases without Diverges and (explicit) Signals

The specification case we are now looking into is roughly the same as in the previous paragraph except that it allows abrupt termination by an Exception:

```java
/*@ public behavior
   @ requires A;
   @ ensures B;
   @ diverges false;
   @ signals (Exception) true;
}@*/
```

which one could ensugar to a behavior specification case:

```java
/*@ public behavior
   @ requires A;
   @ ensures B;
}@*/
```

The meaning of this specification is as follows: If the precondition $A$ and the invariant for type ContextType hold in the prestate, then

- if $m$ terminates normally, the postcondition $m$ holds
- if $m$ terminates by throwing an exception, the exceptional postcondition specified for Exception by the signals clause, which is in our case true, holds.
or expressed in JavaDL:

\[
\forall s : \text{ContextType} \cdot (\{s := s\})
\]

\[
\begin{align*}
Q_{\text{rep}} & Q_{\text{param}} \tau_{\text{update}} (\text{inv} \land [A] \land \text{pre_impl} \rightarrow \\
& \{ \text{java.lang.Exception exc} = \text{null}; \\
& \text{try} \\
& \quad \text{ContextType(result)} :: \text{self.m(p1, ..., p_n)}; \\
& \quad \text{catch(Exception e)} \{ \\
& \quad \quad \text{exc} = \text{e}; \\
& \quad \} \\
& \} Q_{\text{rep}} (\\n& \quad ((\text{exc} = \text{null} \rightarrow [B]) \land \\
& \quad (\text{exc} \neq \text{null} \rightarrow \\
& \quad \quad ((\text{exc instanceof java.lang.Exception} \rightarrow \text{true})))))
\end{align*}
\]

This direct translation result can be simplified, as one easily realizes:

\[
\forall s : \text{ContextType} \cdot (\{s := s\})
\]

\[
\begin{align*}
Q_{\text{rep}} & Q_{\text{param}} \tau_{\text{update}} (\text{inv} \land [A] \land \text{pre_impl} \rightarrow \\
& \{ \text{java.lang.Exception exc} = \text{null}; \\
& \text{try} \\
& \quad \text{ContextType(result)} :: \text{self.m(p1, ..., p_n)}; \\
& \quad \text{catch(Exception e)} \{ \\
& \quad \quad \text{exc} = \text{e}; \\
& \quad \} \\
& \} Q_{\text{rep}} (\text{exc} = \text{null} \rightarrow [B])
\end{align*}
\]

Where \( Q_{\text{rep}}, Q_{\text{param}} \) and \( \tau_{\text{update}} \) have the same meaning as in the previous paragraph.

Here is a short motivation for the formulas above: If \( \text{inv} \land [A] \) holds in the beginning and

- if \( m \) terminates normally, \( \text{exc} \) keeps its initially assigned value \( \text{null} \). In this case \( (\text{exc} := \text{null})(\text{exc} = \text{null} \rightarrow [B]) \) can be simplified to \( [B] \), so the formula states that after a normal termination of \( m \) \( [B] \) holds, which is what we want.

- if \( m \) throws an exception, this exception is caught and assigned to \( \text{exc} \), which is consequently can’t be equal to \( \text{null} \) after that. Therefore \( \text{exc} = \text{null} \rightarrow [B] \) is equivalent to \( \text{true} \) which is just what the \( \text{signals} \) clause specifies for this case.

- if \( m \) loops forever or terminates by throwing an \( \text{java.lang.Error} \), which is not caught by this catch block, the formula is equivalent to \( \text{false} \), which is consistent with our \( \text{diverges} \) clause \( \text{diverges false} \); that doesn’t allow nontermination.

### 5.1.3 Behavior Specifications with Signals Clauses

We extend the previous specification by \( n \) \( \text{signals} \) clauses for the exception types \( E_1, \ldots, E_n \):
5.1. PROVING METHOD SPECIFICATION CASES

/*@ public behavior
@ requires A;
@ ensures B;
@ diverges false;
@ signals (E1 e1) P1(e1);
... 
@ signals (En en) Pn(en);
@*/

This changes the semantics of the specification (compared to the specification in section 5.1.2) in the case of an abrupt termination in the following way:

If \( m \) throws an exception and its type is \( E_i \) for \( 1 \leq i \leq n \) then \( P_i(e_i) \) holds after the exception has been thrown\(^3\).

This specification is translated to:

\[
\forall s : \text{ContextType} \quad \{ \text{self} := s \} \{
Q_{\text{rep}} Q_{\text{param update}} (inv \land \neg A) \land \text{preimpl} \rightarrow
\{ \text{java.lang.Exception exc = null;}
\text{try}\{
\text{ContextType(result) := self.m(p_1, ..., p_n);}
\text{catch(Exception e)\{}
\text{exc = e;}
\}\}
\)Q_{\text{rep}}(
\{(\text{exc = null } \rightarrow \neg B) \land
(\text{exc} \neq \text{null } \rightarrow
((\text{exc instanceof } E_1) \rightarrow \{ e_1 := \text{exc} \} [P_1(e_1)]
\land ((\text{exc instanceof } E_2) \rightarrow \{ e_2 := \text{exc} \} [P_2(e_2)]
\land ...
\land ((\text{exc instanceof } E_n) \rightarrow \{ e_n := \text{exc} \} [P_n(e_n)]
\})
\})
\})
\]

If \( m \) throws an exception, then \( \text{exc} \neq \text{null} \) holds and \( \text{exc} = \text{null } \rightarrow \neg B \) is equivalent to \( \text{true} \). Thus the "postterm" is equivalent to

\[
\bigwedge_{i=1}^{n} (\text{exc instanceof } E_i) \rightarrow \{ e_i := \text{exc} \} [P_i(e_i)].
\]

If the exception's type is \( E_i \) (for \( 1 \leq i \leq n \)), this formula is equivalent to \( \{ e_i := \text{exc} \} [P_i(e_i)] \) otherwise it is equivalent to \( \text{true} \). This matches the semantics of multiple signals clauses.

Remark 5.1.2 (Signals without Parameters) For a signals clauses of the form \( \text{signals } (E) P \) the resulting subterms would be:

\[
(\text{exc instanceof } E) \rightarrow [P]
\]

There are no other effects on the formula above.

\(^3\)Note that the method is not obliged to only being able to throw Exceptions that are instances of \( E_1, \ldots, E_n \). If an exception is thrown whose type matches none of the types specified by the signals clauses, the default exceptional postcondition \( \text{true} \) applies.
5.1.4 Behavior Specifications with Signals and Diverges Clauses

We now consider a fully specified behavior specification which in particular contains a non-trivial diverges clause:

```java
/*@ public behavior
@ requires A;
@ ensures B;
@ diverges C;
@ signals (E1 e1) P1(e1);
...
@ signals (En en) Pn(en);
@*/
```

This changes the semantics of the previous specification case to that effect that `m` may not return to its caller. If that happens, `C` holds in the prestate. This leads to two considerations:

1. If `A` and `!C` hold the method must terminate (either normally or by throwing an `exception`).
2. If `A` holds `B` must hold, if `m` terminates normally, and the exceptional post-conditions specified by the signals clauses must hold if `m` throws an exception. This is basically the same situation as in the previous paragraph. The only difference is that we can’t be sure anymore that `m` terminates. Therefore we get nearly the same formula as this case, but with a box-instead of a diamond modality.

Based on this considerations we get the formula:

```latex
\forall s : [\text{ContextType}] (\{\text{self} := s\} (\text{Q}_r \rho) \text{Q}_r \rho \text{param} \text{update}(\text{inv}) \wedge A \wedge preimpl \rightarrow \\
\begin{array}{c}
\text{prog} \\
\text{Q}_r \rho (\langle (\text{exc} = \text{null} \rightarrow [B]) \wedge \\
(\text{exc} \neq \text{null} \rightarrow \\
\bigwedge_{i=1}^{n} (\text{exc instanceof E}_i) \rightarrow \{e_i := \text{exc}\} [P_i(e_i)] \\
}) \rangle)
\end{array} \\
\wedge (\neg C \rightarrow (\text{prog}\text{true})
```

where `prog` stands for:

```java
java.lang.Exception exc = null;
try{
    ContextType(result) :: self.m(p1,...,p_n);
}catch(Exception e){
    exc = e;
}
```
5.2 Proving Invariants and Constraints

According to [Ref04, p.39 and p.43] a nonhelper method is only obliged to preserve invariants and respect its constraints if the precondition of one of its specification cases holds.

Let \( \text{inv} \) be defined as it is in the previous sections and \( \text{con} \) be the conjoint constraints of type \( \text{ContextType} \), \( \text{m} \) must respect. Then our proof obligation is:

\[
\forall s : [\text{ContextType}] \{ [self := s](Q_{rep}Q_{\text{param}}\tau_{\text{update}}(\negtrail{\text{inv} \land \bigvee_{i=1}^{n} [A_i] \land \text{preimpl} \rightarrow \text{prog}(Q_{rep}(\text{inv} \land \text{con}))))}
\]

Where

- \( \text{prog} \) stands for:
  ```java
  java.lang.Exception exc = null;
  try{
      ContextType(result)::self.m(p1, ..., pn);
  }catch(Exception e){
      exc = e;
  }
  ```
- \( [A_i] \) are the translated preconditions of the method specification cases of \( \text{m} \).

Note that due to the form of \( \text{inv} \) (see section 5.1.1) this formula is also reflecting the fact, that the invariants of objects newly created during the execution of \( \text{m} \) are established.

**Remark 5.2.1 (Invariants and visible states)** JML demands that a nonhelper method has to ensure that the invariants also hold in all visible states during its execution (see [Ref04][p.37]), which includes before the call of other nonhelper methods.

This permits a higher degree of modularity in the verification process, since method calls can then always be eliminated using the corresponding specification\(^4\).

The proof obligation above cannot be used to check this property. It only expresses that the invariants hold after the execution of the regarded method, if they had held in its prestate.

**Remark 5.2.2 (Simplified invariant proof obligation and completeness)**

As for method specification cases (see also remark 5.1.1) it is also possible for invariants to create proof obligations that only express that a method preserves/establishes the instance invariants of the object it belongs to and the static invariants of the type it is declared in.

In this case the following formula would serve as proof obligation:

\[
\forall s : [\text{ContextType}] \{ [self := s](Q_{rep}Q_{\text{param}}\tau_{\text{update}}(\negtrail{\text{inv}_\text{pre}} \land \bigvee_{i=1}^{n} [A_i] \land \text{preimpl} \rightarrow \text{prog}(Q_{rep}([\text{inv}_\text{post}] \land \text{con}))))}
\]

\(^4\) provided that the specification of a method is sufficient, which means describing all necessary aspects of its behavior.
This formula only differs from the "standard" proof obligation to that effect that instead of \( \text{inv} \) the terms \( \text{inv}_{\text{pre}} \) and \( \text{inv}_{\text{post}} \) are added to the pre- and post-condition. \( \text{inv}_{\text{pre}} \) stands for the applicable invariants of type \( \text{ContextType} \) which can be assumed in the prestate of \( m \), whereas \( \text{inv}_{\text{post}} \) stands for the invariants \( m \) has to preserve or establish\(^5\).

Apart from the fact that they can only be used to prove that a method preserves the invariants of the very object/class it belongs to, which makes them less expressive than the standard invariant proof obligation, we lose completeness (see also remark 5.1.1) when using those simplified proof obligations.

This is due to the semantics of JML which only requires that a method meets its specification if the invariants of every existing object/type (if already initialized) hold in its prestate (see also [Ref04, p.51]).

In order to get more modular proof obligations one could also think of providing a proof obligation to check whether a method \( m \) preserves all invariants of a certain type \( T \) or a separate proof obligation for constraints\(^6\).

5.3 Proof Obligations for the Complete Correctness of Method Contracts

For replacing methodcalls by their specification [Ka03] we want to use the complete information that can be derived from a method specification case and not only some aspects of it.

Disregarding details such as the \&pre operator\(^7\) and modifier sets the lemma rules described in [Ka03] have roughly spoken the following form:

Let \( \phi_{\text{pre}} \rightarrow (p)\phi_{\text{post}} \) be a tautology; then the corresponding lemma rule is:

\[
\Gamma \vdash \phi_{\text{pre}}, \phi_{\text{post}}, \Gamma \vdash \psi
\]

\[
\Gamma \vdash (p)\psi
\]

Applied to our situation this means that if we want to describe the complete method contract given by a specification case for \( m \)

- \( \phi_{\text{pre}} \) is the conjunction of the corresponding invariants we can assume to hold in the prestate of \( m \) and the precondition specified by the speccase.

- \( \phi_{\text{post}} \) is the conjunction of the corresponding invariants \( m \) has to preserve or establish, the constraints it has to respect and the postconditions specified by the speccase.

This corresponds to the semantics of a behavior specification case as described in [Ref04, p.51]. Because of the form of \( \phi_{\text{post}} \), that is stronger\(^8\) than the postcondition used in the proof obligations for proving behavior specification cases, a small adaptation of the proof obligation for behavior specification cases is necessary:

Let \( \text{inv} \) and \( \text{con} \) be defined as in the preceding section and

\(^5\)Usually we have \( \text{inv}_{\text{post}} = \text{inv}_{\text{pre}} \) except for a constructor, which cannot assume the instance invariants of the object it creates to hold in its prestate, but has to establish them in its poststate, and the static initialization, which cannot assume the the static invariants of the class it initializes but has to establish them.

\(^6\)These kinds of proof obligations are not implemented yet.

\(^7\)or the \&old operator in our case.

\(^8\)considering the form of the generated rules, it would be a disadvantage to have a postcondition that is weaker than necessary.
5.3. PROOF OBLIGATIONS FOR THE COMPLETE CORRECTNESS OF METHOD CONTRACTS

/*@ public behavior
  requires A;
  ensures B;
  diverges false;
  signals (E1 e1) P1(e1);
  ...
  signals (En en) Pn(en);
  @*/

be a specification case for m. Then the proof obligation that reflects the complete semantics of this specification case is:

$$\forall s : \{\text{ContextType}\} \ (\{\text{self} := s\}$$
$$\begin{align*}
&Q_{rep} Q_{\text{param}}_{\text{update}}(inv \land \top \land pre_{impl} \rightarrow \\
&\begin{cases}
\text{java.lang.Exception exc = null;} \\
\text{try} \\
\quad \text{ContextType(result) :: self.m(p1, ..., p_n);} \\
\quad \text{catch(Exception e)} \\
\quad \quad \text{exc = e;}
\end{cases} \\
\begin{cases}
\text{exc = null } \rightarrow \ [B] \land \\
\text{exc }\neq \text{ null } \rightarrow \\
\quad \wedge \limits_{i=1}^{n} (\text{exc instanceof E}_{i}) \rightarrow \{e_{i} := \text{exc} \} [P_{i}(e_{i})] \\
\end{cases} \\
\wedge inv \land con)
\end{align*}$$

)}}
Chapter 6

Implementation

The translation of JML specifications and the related generation of proofobligations has been implemented and is integrated into the KeY-System.

The java part of JML specifications is parsed by the Recoder [Rec] parser which is already used on other occasions in the KeY-System. The JML annotations themselves are parsed by a parser which was written by using the ANTLR parser generator.

It is possible to create taclets from JML specifications (see [Ka03]) for eliminating method calls. This makes use of the existing taclet creation mechanism for method contracts.

Selecting a proof obligation or displaying the existing specifications is done with the help of a dialogue (as shown in figure 6.1) which can be started from the GUI of the KeY prover.
The input is provided in the form of JML annotated java code (*.java files). Reading specifications from a separate file (see [LBR03, p.9]) is not possible right now.
Chapter 7

Related Works and Tools

A detailed overview of JML tools and information on JML in general can also be found on the website http://jmlspecs.org. For a summary of existing JML tools and their features see [Tools03].

7.1 The Translation from OCL Constraints to JavaDL

Uwe Keller’s diploma thesis [Ke02] gives deeper insights in the translation of OCL constraints, which is part of the KeY system and allows it to combine KeY with CASE-tools based on UML/OCL, such as TogetherCC.

7.2 Runtime assertion Checking with jmlc

The JML compiler (jmlc), which is distributed with the JML tool suite available on http://jmlspecs.org, was developed at Iowa State University. It is an extension to a Java compiler and compiles Java programs annotated with JML specifications into Java bytecode. The resulting bytecode contains assertions for checking pre- and postconditions, invariants, constraints and other specifications. The semantics of the program is not changed as JML specifications are not allowed to have side-effects.

7.3 Unit Testing with jmlunit

jmlunit generates unit tests for the unit testing tool JUnit for Java from JML specifications. The JML specifications thereby serve as a test oracle. The generated testing classes send messages to the tested code and catch assertion violations, based on which it decides whether a test failure has happened or not.

Providing the necessary test data is left to the user.

Though unit tests can be a useful tool for finding bugs and testing the behavior of programs for certain inputs, they do not guarantee the correctness of a program in any way, whereas this is possible by using theorem prover such as KeY.
7.4 Invariant detection with Daikon

Daikon detects likely invariants based on the runtime behavior of a program. Those invariants are pure assumptions and do not claim to be correct, nevertheless they can give useful hints on the properties of the code.

This follows the approach of ESC/Java, i.e. refraining from being sound or complete for the sake of minimal user interaction and a maximum of useful (though sometimes incorrect) results.

7.5 Static Checking with ESC/Java

ESC/Java is a static checking tool developed at Compaq Research that is neither sound nor complete. Besides performing typechecking it can detect common potential errors such as NullPointerException or ArrayIndexOutOfBoundsExceptions. A big advantage of ESC/Java is that it runs fully automatically.

JML annotations can help to suppress unnecessary warnings. For example adding p! = null to the precondition of a method prohibits ESC/Java from reporting potential errors which would occur if p was null.

ESC/Java works by translating a given JML-annotated Java program into verification conditions, which are then passed to an automatic theorem prover. As already mentioned ESC/Java is in contrast to KeY neither sound nor correct.

7.6 Checking Assignable Clauses with Chase

Chase statically checks assignable clauses on a syntactical level. In contrast to the checking of modifier sets implemented in KeY [Sa04] Chase is unsound.

7.7 Program Verification with LOOP

Of the presented tools LOOP is probably the one that is most similar to the KeY approach.

LOOP translates JML-annotated Java code into proof obligations for the theorem prover PVS, which then can be proved interactively. It is hard to give a statement about the advantages and disadvantages of LOOP compared to KeY as it is not publicly available.

7.8 The Krakatoa Tool

Krakatoa translates JML-specified code into proof obligations for the theorem prover Coq. Like KeY, Krakatoa only covers a subset of JML which differs from KeYJML to that effect that Krakatoa has support for loop invariants but apparently not for diverges, history constraints and some special JML constructs like generalized quantifiers.
7.9 JACK

Like Krakatoa and LOOP, the JACK tool generates proof obligation for a theorem prover from JML-annotated Java code. The prover currently used for this purpose is the B method’s prover [Ab96].

Unlike LOOP, JACK aims at being usable for normal Java developers with no experience with formal methods or in the use of theorem provers, which is also a goal of the KeY project.
8.1 Summary

This minor thesis defined the extend of KeYJML. It explained the JML features contained in KeYJML, showed how they are represented in JavaDL and tried to give an intention for this in cases it was not obvious. In this context it was tried to match the JML semantics as described in [Ref04] as closely as possible, though as far as the issue of Errors and how they are related to diverges is concerned the KeYJML semantics follows the approach of [LBR03, p.61].

We have seen that most JML expressions, since they often syntactically and semantically correspond to Java expressions, can be translated to their JavaDL counterparts quite canonically.

This thesis also gave an overview of various kinds of proof obligations that can be created to prove certain aspects of the translated specifications. The proof obligations and the translation results meet the quality criteria presented in section 1.4 insofar as it’s transparent which part of the formula corresponds to which aspect of the specification.

Queries were used several times in order to make the translation results of JML expressions as simple as possible. This usually stood in contrast to the criterion of optimal machine usability, as it caused additional proof steps for resolving the query. But as the amount of additional rule applications was justifiable, in cases of the absence of a feasible alternative, the decision was made in favour of using queries.

8.2 Outlook

Although KeYJML is already rather expressive, compared with what is used in real-world specifications, there are various extension possibilities for it. Loop invariants for instance are not supported at the moment, but surely would be quite useful to have in some cases. As it should be no problem, from a technical point of view, to implement support for loop invariants, this is one sensible extension possibility for the near future.

Being able to handle expressions like a[+] or a[b..c] would also be nice. But this rather touches the handling modifier sets within KeY in general than being a mere JML translation issue. Another heavy restriction concerning modifier sets
is the lacking support for datagroups which would be needed for using model fields in assignable clauses.

Another issue one could tackle sooner or later is the translation of special JML expressions such as \texttt{\textbackslash fresh} or \texttt{\textbackslash not\_modified}. \texttt{\textbackslash not\_modified(a_1,\ldots,a_n)} for instance can be translated to:

\[
\bigwedge_{i=1}^{n} F(a_i)
\]

where \(F(a_i)\) is

\[
F(a_i) := \begin{cases} 
\text{if the type of } a_i \text{ is primitive} \\
[a_i =\!\!\!\!\!\!\text{\texttt{\textbackslash old}(a_i)}] \\
(a_i =\!\!\!\!\!\!\text{\texttt{null} } \rightarrow a_i =\!\!\!\!\!\!\text{\texttt{\textbackslash old}(a_i)}}) \land \\
(a_i \neq \text{\texttt{null} } \rightarrow a_i\text{.equals(\text{\texttt{\textbackslash old}(a_i)}})) \\
\text{otherwise}
\end{cases}
\]

\texttt{\textbackslash fresh(a_1,\ldots,a_n)} means, that \texttt{a_1,\ldots,a_n} are not \texttt{null} and the objects they refer to were not allocated in the pre-state.

This section should just give some examples for existing restrictions of KeY-JML and the possibilities to remove them. Which features will really be added to KeYJML in the end will of course heavily depend on what turns out to be really needed and which new JML features will emerge in the future, as JML itself is still under development and its language definition has not reached a stable state yet.
Appendix A

Syntax of KeYJML

The syntax of KeYJML is described as an EBNF grammar, where

- nonterminals are enclosed by < and >.
- groups are enclosed by braces.
- ( A )* denotes a sequence of 0 or more As.
- ( A )+ denotes a sequence of 1 or more As.
- elements enclosed by square brackets are optional (0 or 1 time).
- if a terminal t can be confused with an EBNF symbol it is written as ‘t’.

A.1 Field Syntax

<field> ::= <modifiers> (<member-decl> | <jml-declaration>)
<modifiers> ::= ( <modifiers> )*
<modifier> ::= public | protected
    | private | instance
    | static | model
    | pure | helper
<member-decl> ::= <method-decl>
    | <variable-definition>
<variable-decls> ::= <type-spec> <variable-declarators> ;
<type-spec> ::= <type> '[' <dims> ']'    
<type> ::= <reference-type> | <builtInType>
<reference-type> ::= <name>
<name> ::= <ident> ( . <ident> )*
<builtInType> ::= void | boolean
    | byte | short
    | int | long
<variable-declarators> ::= ( <variable-declarator> )+
<variable-declarator> ::= <ident> [ <dims> ]
<method-decl> ::= <method-specification>
    <modifiers> [ <type-spec> ] <method-head>
APPENDIX A. SYNTAX OF KEYJML

<method-head> ::= <ident> ‘(’ [ <param-declaration-list> ] ‘)’
<param-declaration-list> ::= <param-declaration> ( , <param-declaration> )*
<param-declaration> ::= <type-spec> <ident> [ <dims> ]
<dim> ::= ( ‘[ ’ ’] ’ )+

A.2 Behavioral Specification Syntax for Types

<jml-declaration> ::= <invariant> | <history-constraint>
               | <represents-decl>
<invariant> ::= <invariant-keyword> <predicate> ;
<invariant-keyword> ::= invariant | invariant_redundantly
<history-constraint> ::= <constraint-keyword> <predicate>
<constraint-keyword> ::= constraint | constraint_redundantly
<represents-decl> ::= <represents-keyword>
              | <represents-keyword> <store-ref-expression> <l_arrow-or-eq> <spec-expression> ;
<represents-keyword> ::= represents | represents_redundantly
<l_arrow-or-eq> ::= <- | =

A.3 Store Ref Syntax

<store-ref-list> ::= <store-ref> ( , <store-ref> )*
<store-ref> ::= <store-ref-expression>
               | <store-ref-keyword>
<store-ref-expression> ::= <store-ref-name> ( <store-ref-name-suffix> )*
<store-ref-name> ::= <ident> | this
<store-ref-name-suffix> ::= . <ident> | . this
                        | ‘[ ’ spec-array-ref-expr ‘] ’
<spec-array-ref-expr> ::= <spec-expression>
<store-ref-keyword> ::= \nothing | \everything | \not_specified

A.4 Behavioral Specification Syntax for Methods

<method-specification> ::= <specification>
                        | <extending-specification>
<specification> ::= <spec-case-seq>
<spec-case-seq> ::= <spec-case> ( also <spec-case> )*
<spec-case> ::= <lightweight-spec-case>
               | <heavyweight-spec-case>
<lightweight-spec-case> ::= <generic-spec-case>
<extending-specification> ::= also <specification>
A.4. BEHAVIORAL SPECIFICATION SYNTAX FOR METHODS

<privacy> ::= public | protected | private
<exceptional-simple-spec-body> ::= ( <exceptional-spec-clause> )+
<exceptional-spec-clause> ::= <diverges-clause>
| <assignable-clause> | <signals-clause>
<normal-simple-spec-body> ::= ( <normal-spec-clause> )+
<normal-spec-clause> ::= <diverges-clause> | <assignable-clause>
| <ensures-clause>

A.4.1 Lightweight Specification Cases
<generic-spec-case> ::= <spec-header> [ <generic-spec-body> ]
| <generic-spec-body>
<spec-header> ::= ( <requires-clause> )+
<generic-spec-body> ::= <simple-spec-body>
| ‘{|’ <generic-spec-case-seq> ‘|}’
<generic-spec-case-seq> ::= <generic-spec-case> ( also <generic-spec-case> )*
<simple-spec-body> ::= ( <simple-spec-body-clause> )+
<simple-spec-body-clause> ::= <diverges-clause> | <assignable-clause>
| <ensures-clause> | <signals-clause>

A.4.2 Heavyweight Specification Cases
<heavyweight-spec-case> ::= [ [ <privacy> ] behavior <generic-spec-case>
| [ [ <privacy> ] exceptional_behavior
<exceptional-spec-case>
| [ [ <privacy> ] normal_behavior <normal-spec-case>
<exceptional-spec-case> ::= <spec-header> [ exceptional-spec-body ]
| <exceptional-spec-body>
<exceptional-spec-body> ::= ( <exceptional-spec-clause> )+
| ‘{|’ <exceptional-spec-case-seq> ‘|}’
<exceptional-spec-case-seq> ::= ( <exceptional-spec-case> )+
<normal-spec-case> ::= <spec-header> [ <normal-spec-body> ]
| <normal-spec-body>
<normal-spec-body> ::= ( <normal-spec-clause> )+
| ‘{|’ <normal-spec-case-seq> ‘|}’
<normal-spec-case-seq> ::= ( <normal-spec-case> )+

A.4.3 Method Specification Clause Syntax
<requires-clause> ::= <requires-keyword> <pred-or-not> ;
<requires-keyword> ::= requires | pre
| requires_redundantly | pre_redundantly
<pred-or-not> ::= <predicate> | \not_specified
<assignable-clause> ::= <assignable-keyword> <conditional-store-ref-list> ;
<assignable-keyword> ::= assignable | assignable_redundantly
APPENDIX A. SYNTAX OF KEYJML

| modifiable | modifiable_redundantly
| modifies | modifies_redundantly

<conditional-store-ref-list> ::= ( <conditional-store-ref> )+
<conditional-store-ref> ::= <store-ref>
<ensures-clause> ::= <ensures-keyword> <pred-or-not> ;
<ensures-keyword> ::= ensures | post
| ensures_redundantly | post_redundantly
<signals-clause> ::= <signals-keyword>
  (' <reference-type> [ <ident> ] ')' <pred-or-not> ;
<signals-keyword> ::= signals | signals_redundantly
| ensures | ensures_redundantly
<diverges-clause> ::= <diverges-keyword> <pred-or-not> ;
<diverges-keyword> ::= diverges | diverges_redundantly

A.5 Predicate and Specification Expression Syntax

<predicate> ::= <spec-expression>
<spec-expression-list> ::= ( <spec-expression> )+
<spec-expression> ::= <expression>
<jml-primary> ::= \result
| \old ( '\ <spec-expression> ')
| \nonnullelements ( '\ <spec-expression> ')
| <spec-quantified-expri
<spec-quantified-expri ::= '\ <quantifier> <quantified-var-decls> ;
  [ [ <predicate> ] ; ] <spec-expression> ')
<quantifier> ::= \forall | \exists | \max | \min
<quantified-var-decls> ::=<type-spec> <quantified-var-declarator>
  ( , <quantified-var-declarator> )*<quantified-var-declarator> ::= <ident> [ <dims> ]

A.6 Expression Syntax

<expression-list> ::= ( <expression> )+
<expression> ::= <assignment-expr>
<assignment-expr> ::= <conditional-expr>
<conditional-expr> ::= <equivalence-expr>
[ ? <conditional-expr> : <conditional-expr> ]
<equivalence-expr> ::=<implies-expr> ( <equivalence-op> <implies-expr> )*<equivalence-op> ::= =<implies-expr> ::= <logical-or-expr>
[ ==> <implies-non-backward-expr> ]
| <logical-or-expr> <= <logical-or-expr>
( <= <logical-or-expr> )*<implies-non-backward-expr> ::= <logical-or-expr>
A.7 Tokens

[ ==> <implies-non-backward-expr> ]

<logical-or-expr> ::= 
    <logical-and-expr> ( `||' <logical-and-expr> )* 
<logical-and-expr> ::= 
    <inclusive-or-expr> ( && <inclusive-or-expr> )* 
<inclusive-or-expr> ::= 
    <exclusive-or-expr> ( `|' <exclusive-or-expr> )* 
<exclusive-or-expr> ::= <and-expr> ( `^' <and-expr> )* 
<and-expr> ::= <equality-expr> ( & <equality-expr> )* 
<equality-expr> ::= <relational-expr> [ == <equality-expr> ] 
    | <relational-expr> [ != <equality-expr> ] 
<relational-expr> ::= <shift-expr> < <shift-expr> 
    | <shift-expr> > <shift-expr> 
    | <shift-expr> <= <shift-expr> 
    | <shift-expr> >= <shift-expr> 
    | <shift-expr> [ instanceof <type-spec> ] 
<shift-expr> ::= <additive-expr> ( <shift-op> additive-expr )* 
<shift-op> ::= << | >> | >>>
<additive-expr> ::= <mult-expr> ( <additive-op> mult-expr )* 
<additive-op> ::= + | - 
<mult-expr> ::= <unary-expr> ( <mult-op> unary-expr )* 
<mult-op> ::= * | / | % 
<unary-expr> ::= + <unary-expr> 
    | - <unary-expr> 
    | <unary-expr-not-plus-minus> 
<unary-expr-not-plus-minus> ::= "-" <unary-expr>
    | ! <unary-expr> 
    | '(' <builtinType> ')’ <unary-expr> 
    | <postfix-expr> 
<postfix-expr> ::= <primary-expr> ( <primary-suffix> )* 
<primary-suffix> ::= . <ident> 
    | ‘(’ [ <expression-list> ] ‘)’ 
    | ‘[’ <expression> ‘]’ 
<primary-expr> ::= <ident> | <constant> 
    | true | false | this 
    | null | ‘(’ <expression> ‘)’ 
    | <jml-primary> 
<builtinType> ::= void | boolean | byte 
    | char | short | int 
    | long 
<constant> ::= <java-literal> 
<dims> ::= ( ‘[’ ‘]’ )+
<java-literal> ::= <integer-literal>
<integer-literal> ::= <decimal-integer-literal>
<decimal-integer-literal> ::= <decimal-numeral>
<decimal-numeral> ::= 0 | <non-zero-digit> [ <digits> ]
<digits> ::= ( <digit> )+ 
<digit> ::= 0 | <non-zero-digit>

<non-zero-digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<keyword> ::= <java-keyword> | <jml-predicate-keyword> | <jml-keyword>
<jml-predicate-keyword> ::= \duration | \elemtype | \forall | \exists | \一切 | \最
<jml-keyword> ::= abrupt_behavior | accessible_redundantly | accessible | also | assert_redundantly | assignable_redundantly | assignable | assume_redundantly | assume | axiom | behavior | breaks_redundantly | breaks | callable_redundantly | callable | choose_if | choose | code_contract | constraint_redundantly | constraint | constructor | continues_redundantly | continues | decreases_redundantly | decreases | decreasing_redundantly | decreasing | diverges_redundantly | diverges | duration_redundantly | duration | ensures_redundantly | ensures | example | exceptional_behavior | exceptional_example | ensures_redundantly | ensures | field | forall | for_example | ghost | helper | hence_by_redundantly | hence_by | implies_that | in | initializer | initially | instance | invariant_redundantly | invariant | loop_invariant_redundantly | loop_invariant | maintaining_redundantly | maintaining | maps | measured_by_redundantly | measured_by | method | model_program | model | modifiable_redundantly | modifiable modifies_redundantly | modifies | monitored_by | monitored | non_null | normal_behavior | normal_example | nowarn | old | or | post | pre
Index

access modifiers, 6
additive expressions, 33
and
    bitwise, 32
    logical, 30
ANTLR, 50
assignable, 17
behavior, 13
boolean
    arguments of methods, 40
    expressions, 6
Chase, 53
conditional expressions, 28
constraints, 21
Daikon, 53
datagroups, 18
diverges, 14
    proof obligation, 46
ensures, 14
    proof obligation, 43
equality expressions, 32
equivalence expressions, 29
errors, 7, 14
Esc/Java, 53
exceptional_behavior, 13
exclusive or expressions, 31
exists, 34
expressions, 28
forall, 34
forall, 34
heavyweight specification cases, 12
    proof obligation, 43
helper, 19
implies expressions, 29
inheritance
    constraints, 21
    limits, 18
    parameters, 39
invariants, 21
method specifications, 19
model fields, 25
represents, 25
invariants, 20
Jack, 54
JavaDL, 1
JML Specification Browser, 50
jmlc, 52
jmlunit, 52
KeYJML, 10
    syntax, 57
Krakatoa, 53
lightweight specification cases, 12
Loop, 53
max, 34
method specifications, 11
    defaults, 11
    inheritance, 19
min, 34
model fields, 22
    inheritance, 25
model methods, 21
modifies, 17
multiplicative expressions, 33
nested method specification, 13
\nonnullelements, 35
normal behavior
    proof obligation, 42
normal_behavior, 13
\old, 37
or
    bitwise, 30
    logical, 30
parameters, 39
privacy modifiers, 6
proof obligations, 41
pure, 18

quality criteria, 4
quantifiers, 34
query, 2

Recoder, 50
redundant specifications, 8
relational expressions, 33
requires, 14
\result, 38
rigid
terms, 2
variables, 2

shift expressions, 33
signals, 15
proof obligation, 44

this, 39
types
translation, 27

unary expressions, 33

variables
rigid and nonrigid, 2
translation, 39
visible states, 47
Bibliography


[ANTLR] www.antlr.org The ANTLR homepage


[Bec01] B. Beckert, A Dynamic Logic for the formal verification of Java Card programs. In Java on Smart Cards: Programming and security. Revised Papers, Java Card 2000, International Workshop, Cannes, 2000

[BP03] C.-B. Breunesse, E. Poll, Verifying JML specifications with model fields, Department of Computer Science, University of Nijmegen, 2003

[BS03] B. Beckert, P. H. Schmitt, Program Verification Using Change Information, University of Karlsruhe, Department of Computer Science, 2003


[KeY] www.key-project.org, The KeY homepage

[Hu00] M. Huisman, Reasoning about Java Programs in higher order logic with PVS and Isabelle In IPA dissertation series, 2001-03, University of Nijmegen, Holland, February 2001.


[LC04] G. T. Leavens, Y. Cheon, Design by Contract with JML


[Rec] recoder.sourceforge.net the RECODER homepage


[Ro02] A. Roth, Deduktiver Softwareentwurf am Beispiel des Java Collections Frameworks, Diploma Thesis, University of Karlsruhe, Department of Computer Science, June 2002
