Karlsruhe Institute of Technology

Proving Well-Definedness of JML Specifications with KeY

Study Thesis by

Michael Kirsten

Department of Informatics
Institute of Theoretical Informatics (ITI)
Logic and Formal Methods

Supervisor: Prof. Dr. rer. nat. Peter Hans Schmitt
Advisors: Dr. rer. nat. Mattias Ulbrich
Dipl.-Inform. Christoph Scheben

November 27, 2013

Abstract

Specification methods in formal program verification enable the enhancement of source code with formal annotations as to formally specify the behaviour of a program. This is a popular way in order to subsequently prove software to be reliable and meet certain requirements, which is crucial for many applications and gains even more importance in modern society. The annotations can be taken as a contract, which then can be verified guaranteeing the specified program element – as a receiver – to fulfil this contract with its caller. However, these functional contracts can be problematic for partial functions, e.g. a division, as certain cases may be *undefined*, as in this example a division by zero. Modern programming languages such as Java handle *undefined* behaviour by casting an exception.

There are several approaches to handle a potential *undefinedness* of specifications. In this thesis, we chose one which automatically generates formal proof obligations ensuring that *undefined* specification expressions will not be evaluated.

Within this work, we elaborate on so-called *Well-Definedness Checks* dealing with *undefinedness* occurring in specifications of the modelling language JML/JML* in the KeY System, which is a formal software development tool providing mechanisms to deductively prove the before mentioned contracts. Advantages and delimitations are discussed and, furthermore, precise definitions as well as a fully functional implementation within KeY are given. Our work covers the major part of the specification elements currently supported by KeY, on the higher level including class invariants, model fields, method contracts, loop statements and block contracts. The process of checking the *well-definedness* of a specification forms a preliminary step before the actual proof and rejects *undefined* specifications. We further contribute by giving a choice between two different semantics, both bearing different advantages and disadvantages. The thesis also includes an extensive case study analysing many examples and measuring the performance of the implemented *Well-Definedness Checks*.

Deutsche Zusammenfassung

Spezifikationstechniken der formalen Programmverifikation erlauben es, Quelltext mit formalen Annotationen zu versehen und damit Programmverhalten formal zu spezifizieren. Dies ist eine weit verbreitete Herangehensweise, um Software anschließend auf Verlässlichkeit und die Erfüllung bestimmter Forderungen zu überprüfen und Programmeigenschaften insbesondere formal zu beweisen. Solche Überprüfungen und Beweise sind für viele Anwendungen äußerst wichtig und nehmen in der modernen Gesellschaft einen wachsenden Stellenwert ein. Formale Annotationen können als ein funktionaler Vertrag oder Kontrakt verstanden werden, die verifiziert werden können und somit zusichern, dass das spezifizierte Programmelement diesen gegenüber seinem Nutzer beziehungsweise Aufrufer erfüllt. Diese funktionalen Kontrakte können in sich aber bereits problematisch sein, wenn sie partielle Funktionen, wie beispielsweise eine Division, spezifizieren. Solche Funktionen besitzen die Eigenschaft, dass bestimmte Fälle, wie eine Division durch Null, *undefiniert* sind. Moderne Programmiersprachen wie Java veranlassen beim Auftreten von undefiniertem Verhalten eine Ausnahmebehandlung, sie werfen eine *Exception*.

Es gibt verschiedene Ansätze, mit der potentiellen Undefiniertheit von Spezifikationen umzugehen. In dieser Studienarbeit wird ein Ansatz gewählt, der automatisch formale Beweisverpflichtungen erzeugt, die sicherstellen, dass in Spezifikationen keine *undefinierten* Ausdrücke zur Auswertung kommen.

Innerhalb dieser Arbeit werden sogenannte *Wohldefiniertheitstests* oder *Well-Definedness Checks* erarbeitet, die mit dem Auftreten von *undefinierten* Ausdrücken in Spezifikationen der Modellierungssprache JML/JML* im KeY System umgehen. KeY ist ein formales Werkzeug der Softwareentwicklung, das Mechanismen zum deduktiven Beweisen der erwähnten Kontrakte bereitstellt. Vor- und Nachteile werden erörtert und des Weiteren präzise Definitionen sowie eine voll funktionale Realisierung innerhalb von KeY gegeben. Diese Studienarbeit deckt den Hauptteil der aktuell von KeY unterstützen Spezifikationselemente ab. Darin sind die übergeordneten Elemente Klasseninvarianten, Modellfelder, Methodenkontrakte, Schleifenanweisungen und Blockverträge enthalten. Das Verfahren, die *Wohldefiniertheit* einer Spezifikation zu prüfen, stellt einen dem eigentlichen Beweis vorangehenden Schritt dar und sortiert *undefinierte* Spezifikationen aus. Zusätzlich werden zwei verschiedene Semantiken – wobei im Programm eine der beiden gewählt werden kann – beigesteuert, die beide verschiedene Vor- und Nachteile aufweisen. Diese Arbeit enthält außerdem eine Evaluierung einer umfangreichen Menge von Fallbeispielen, innerhalb derer viele Beispiele und die Effizienz und Laufzeit der implementierten *Well-Definedness Checks* untersucht werden.

Acknowledgements

I would like to thank Anh Phan Duc and Thomas Schenker for proofreading this thesis, as well as Adam Siders for many grammatical corrections. Furthermore, thanks go to Christoph Scheben for proposing this exciting topic and Prof. Dr. Peter H. Schmitt for giving me the great opportunity to work on this subject and to gain many experiences and insights in the KeY project and the field of formal verification. I'm also very grateful to Prof. Dr. Peter H. Schmitt and Prof. Dr. Bernhard Beckert for enthusing me for the domain of formal logic in the first place. A special thanks goes to my advisors Christoph Scheben and especially Dr. Mattias Ulbrich for taking the time for me, motivating me and many lively debates supporting my work. Last but not least, I want to thank my parents for all the support for me and my studies over the years.

Statement of Authorship

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, November 27, 2013

.....

(**Michael Kirsten**)

Contents

1. Introduction

For computerised systems and with them software being an integral part of modern society, reliability is and becomes more and more indispensable. In particular, applications in medical engineering, transportation and security imply a strong need for reliable software. One major approach to analyse this quality is software verification using formal methods. However, when talking about reliability, it falls into place that a universal idea of reliability seems neither feasible nor possible. Hence, we need to enhance software by a definition of its reliability. To handle this in a formal way, formal specifications defined by a formal specification language are needed. One major concept dealing especially with object-oriented programming languages is *design by contract*, where operations are specified on the basis of pre- and postconditions, and objects by means of invariants. These specifications are contracts between caller and receiver and, if proven correctly, certify a certain behaviour for the specified operation or object.
This thesis examines one modelling language for formal specifications and proposes a way to deal with potentially undefined, ill-defined or meaningless specifications. Our goal can be described as to not give room to undefinedness resulting in potentially unintended or ambiguous specifications. We will perform this on JML*, a variant of the *Java Modeling Language* (JML), by using the KeY System, a formal and deductive software verification tool. The result will be precise formal proof obligations for the most common specification elements and their practical implementation in KeY, incorporating two significantly different theories on the meaning of *well-definedness*.

1.1. Well-Definedness

A rough description of undefinedness could be a "lack of information". To give a simple example, let us take the arithmetic operation *division*. It is not *well-defined* for its divisor being equal to zero. Only with the information that this is not the case, we can guarantee the result to be *defined*. In many programming languages, the occurrence of an *undefined* expression results in an exception at run-time. Since specifications are not part of the execution of a program, they do not play any role at run-time and thus cannot produce any exception at run-time. However, the possibility of throwing an exception at run-time if being executed can very well play a role for its verification and thus a need for an equivalent on specification level comes into being.
Let us further discuss the before mentioned example of the division. Furthermore, a well-known law in classical two-valued logic is the *law of excluded middle* or *tertium non datur*. By making use of this law, most calculi would deduce a formula as e.g. "$x/0 = 1 \lor \neg(x/0 = 1)$" to be valid. However, most compilers for modern programming

languages would cast an exception and thus **not** take it to be valid. At least for the matter of deductive formal program verification, this becomes an issue as both logicians as well as programmers are involved. Besides, the matter is not as trivial as it might seem in this simple example, because it concerns all partial functions and predicates. As mentioned in [BCJ84], especially recursive definitions often result in partial functions and its iteration gives rise to programs which may fail to terminate for some inputs. The work also states that proofs about such functions or programs should be conducted in logical systems which reflect the possibility of "*undefined* values". Moreover, [Sch11] even gives examples that for alternative solutions such as *under-specification* (with the advantage to not need to define evaluation of terms and formulas in partial structures), a lot of non-intuitive or even strange properties can be derived. This rules out the option of ignoring this discrepancy and leaves us with the question of whether to change one of those semantics in favour of the other or to find a different means of uniting them both.

1.2. Related Work

In [AM02], six different approaches dealing with *undefinedness* in program verification are presented and evaluated on the example of the arithmetic formula "$2 + \frac{3}{0} = 2$" and its negation "$2 + \frac{3}{0} \neq 2$". We will briefly review the different possibilities in the following.

1. **Three-Valued Logic:** In this approach, we use the value *undefined* to express undefinedness. Solutions using this approach are sometimes referred to as *weak logic theory* [Hol91]. The general idea is to extend the usual set of truth-values $\{true, false\}$ by the value "$\bot$" denoting *undefined* and also the domain likewise. In some sense, this is the "proper way to do it" as it tackles the appearance of *undefined* in any place of the proof. The main disadvantages are a loss of applicable axioms (e.g. the *law of excluded middle*) and the open question of how to propagate the *undefined* value through functions (e.g. what is the value of "$false \wedge \bot$"?) [Häh05]. Practically speaking, the advantages of this approach do not outweigh the fact that proofs become much harder herein. In practice it might also be useful to find out the source of the *undefinedness* (similar to exception types in programs), so it might be necessary to define more than one value to denote *undefined*, which then results in an unjustified complex semantics.

2. **Forced Value:** In this approach, we always assign a valid value. Here one would, for example, systematically give the value zero to a term of the form $\frac{a}{0}$, which is probably a very unexpected result. Consequently some kind of "zero-value" would be necessary for every result type. We gain from this approach that every value is fixed and we stay in two-valued semantics. The argument against this approach is that "many mathematicians feel uneasy with the identity $\frac{x}{0} = 0$." [Sup57].

3. **Under-Specification:** In this approach, we always claim a term to have a value (in classical logic). However, this value might be *unknown*, i.e. not fixed and without assumptions about it. This approach does not allow any assumptions or constraints about the term $\frac{a}{0}$, but saves equality, i.e. $\frac{a}{0} = \frac{a}{0}$ holds.

4. **Conditional Definitions:** In this approach, we banish any definition of new term symbols and only allow new predicate symbols. Hence, division would be defined by a predicate symbol $\mathrm{Div}(a, b, c)$ meaning $a = \frac{b}{c}$ with the definition "$\mathrm{Div}(a, b, c) \Leftrightarrow \neg\,\mathrm{Zero}(c) \wedge \mathrm{Mult}(b, a, c)$". This has the downside of a rather heavy system of definitions and still leaves *undefined* terms indiscernible.

5. **Evaluation to False:** In this approach, *undefined* predicates always evaluate to *false*. This approach is argued to be the one "commonly used by mathematicians" [AM02]. It leaves us with the problem of not being able to distinguish *undefined* predicates of *defined*, but still invalid predicates. Therefore, the approach highly differs from classical logic.

6. **Rejection:** In this approach, we reject to prove *ill-defined* expressions [BBM98] [DMR08]. This approach is based on the idea that a formal language containing potentially *ill-defined* expressions can be given a semantic interpretation within a three-valued logical domain. In this way, the approach is very similar to the first one using a *three-valued logic*, but the practical treatment of *undefinedness* is different. Here, *undefinedness* does not interfere with two-valued logic as it proposes a *Well-Definedness Check*, which acts as a *filter* before the actual proof. If this check is successful for some formula, i.e. proves it to be *well-defined*, it is guaranteed that the result of the proof can never be *undefined* if evaluated in a *three-valued logic*, leaving us with a *pseudo-two-valued logic*.

To give an understanding of these approaches, table 1.1 illustrates the evaluation of our example:

Formula	$2 + \frac{3}{0} = 2$	$2 + \frac{3}{0} \neq 2$
Appr. 1	undefined	undefined
Appr. 2	true	false
Appr. 3	unprovable	unprovable
Appr. 4	true	true
Appr. 5	false	false
Appr. 6	rejected	rejected

Table 1.1.: Exemplary illustration for the various approaches to treat *undefinedness*

All these approaches treat the issue on a purely logical or mathematical level. In [Sch11], a definition supporting the approaches 1, 3 and 6 is presented:

Definition 1 *A formula ϕ is* well-defined *with respect to a theory* Th *if the truth value of ϕ in any model of* Th *does not depend on the values of functions f outside their fixed value formula* fix_f.

However, since we talk about specification languages aimed and designed to describe and specify program elements, it can be justified to treat the issue on a somehow more "programming" kind of level acting similar to a run-time assertion checker (RAC). This point of view is pursued in [Cha05], arguing that it is mostly programmers who write specifications and thus are accustomed to the semantics of programming languages similar to Java, where *short-circuit evaluation* is a common practise.
Another opinion, especially since we are talking about the specification language JML, is given in [Lea+11]. Initially, JML semantics for *undefinedness* was based on approach 3 [Ulb09], but in 2007, semantics was changed to be based on "strong validity", which is defined as follows:

Definition 2 *An assertion is taken to be valid if and only if its interpretation does not cause an exception to be raised, and yields the value true.*

This would actually support approach 5, with the rather heavy downside of not being able to distinguish *undefined* predicates of *defined*, but still invalid predicates. Thus, considering the views from [Cha05] with respect to the very reasonable definition 1, this leaves us with approaches 1 and 6. To cope with definition 2, we simply evaluate *undefined* results to *false*, but still know the reason for this evaluation.
In the following, we decided for an approach in the spirit of approach 6 by using a *Well-Definedness Operator* (section 2.5). For this operator, we basically present two different options or semantics, one treating *well-definedness* on a purely mathematical level and another one taking the programming level into account.

1.3. Outline

The rest of this work is structured as follows.

- Chapter 2 presents the fundamental terms, concepts and notions used in this thesis. This includes the specification language JML, the extensions and adjustments defined by JML*, a procedure to desugar JML specification elements using the example of method contracts, the KeY System, the *Well-Definedness Operator* and different semantics for it, the black box semantics assumed in this work and the underlying state semantics.

- Chapter 3 formally defines the operator introduced in this thesis, with all its detailed rules for predicates, expressions and whole specification elements as class invariants, model fields and method contracts as well as specification statements such as loop statements and block contracts.

- Chapter 4 describes the implementation of *Well-Definedness Checks* in the KeY System, justifies our design decisions, discusses the rule implementation and illustrates the practical process on two short examples, thereby discovering new *undefinedness* problems.

- Chapter 5 demonstrates the usage of our implementation on a set of examples, discusses specifications found to be *ill-defined* by classifying them according to Java exception types and finally depicts performance observations for the presented *Well-Definedness Check*.
- Chapter 6 finally concludes by discussing the results of this work and giving an outlook on further ideas and future work related to the topic.

2. Fundamentals

In the following, we will give some fundamental concepts, upon which this work is based. More specifically, section 2.1 will introduce the reader with JML, the essential modelling language which defines the specification elements, for which we developed our *Well-Definedness Checks*. To be more precise, we actually used JML*, described in section 2.2. Furthermore, section 2.3 elaborates the process for desugaring these specification elements as to have exact blueprints to talk about. Section 2.4 familiarises the reader with the KeY System, which is the development tool for our practical work. In section 2.5, the fundamental operator for *well-definedness* with two different semantics is given. An assumption for our semantic considerations is given in section 2.6. Finally, section 2.7 sets the stage for a common understanding of our abstract concept of programs, namely state semantics.

2.1. The Java Modeling Language (JML)

JML is a notation for formally specifying the behaviour and interfaces of Java classes and methods [Lea+11], which makes it a formal behavioural interface specification language (BISL). It contains the essential notations used in DBC (*design by contract*) as a subset [LC06]. The principal idea behind DBC is that a class and its clients have a *contract* with each other. The client must guarantee certain conditions before calling a method defined by the class, and in return the class guarantees certain properties that will hold after the call. JML is targeted to provide a comprehensive specification of both *interfaces* (names and static information in Java declarations, i. e. syntax) and *behaviour* (how the module acts when used, i. e. semantics) for every aspect of Java and simultaneously retain an easy-to-read format [LBR06]. As such, it pursuits three major goals [LC06]:

1. "JML must be able to document the interfaces and behaviour of existing software, regardless of the analysis and design methods to create it."
2. "The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training."
3. "The language must be capable of being given a rigorous formal semantics, and must also be amenable to tool support."

It is primarily intended to specify existing code, rather than to implement programs according to a preexisting specification. These specifications are written directly into the source files of Java modules, having the basic syntax of expressions — with some extensions — as in Java. As they are always given inside comments, JML constitutes

a proper extension to the Java language, not interfering with the program execution. JML is being developed and maintained by a community led by Gary T. Leavens at the University of Central Florida (UCF).
It primarily serves the purpose of documenting artefacts of the Java language. Applications of JML include static type checking, run-time assertion checking as well as deductive formal verification methods. Regarding its structure, JML includes a common expression language as almost[1] a superset of Java expressions and — being a higher level — several *specification elements* in order to describe the program behaviour in different ways. These *specification elements* include *assertion statements* within the program code, *method specifications* (based on the DBC paradigm) and *type specifications* such as invariants, which must be preserved throughout the run of a program. Moreover, variables and fields may be used in specifications (model fields, ghost variables/fields).

2.2. JML*

JML* is a variant of JML making use of dynamic frames [Kas06]. The idea consists in decoupling location sets (data groups) from model fields, and to make them first-class elements of the language. Consequently, in JML* data groups and data group inclusions are omitted. It also introduces a new *primitive type* (as `int` or `boolean`), namely `\locset`, standing for sets of memory locations and replacing JML's "store ref expressions" by a proper type. This approach and further features of JML* are covered in-depth in [Wei11].

2.3. Desugared Specifications

JML (or JML* in our case) features a great deal of syntactic sugar that is designed to enrich specifications to be more comprehensible and usable [RL05]. Since dealing with this syntactic sugar would be redundant and there is an algorithm for *desugaring* JML/JML*-specifications, this subsection will describe the format of *desugared* specifications in order to not lose focus and concentrate on the core syntax of specifications (without losing any expressiveness) in the following.
The desugaring process for method specifications as in [RL05] is an algorithm of 11 steps, where $V \in \{\texttt{public}, \texttt{private}, \texttt{protected}\}$ (allowed to be empty) denotes a visibility level. We mostly stick to this algorithm by performing the following steps:

1. Desugar the use of the `non_null` modifier for argument types of reference or array type by explicitly prepending according preconditions, e.g. x `!= null` for x being an argument of reference type, to the precondition of every individual corresponding specification case.

2. Desugar the use of the `non_null` modifier for result types of reference or array type by prepending explicit postconditions to the original postcondition, in a similar fashion as the previous step was performed.

[1] This is except for expressions with side-effects such as e.g. `i++` for `i` being of type `int`.

3. Desugar the use of `pure` by adding `diverges false` (for every specification case) and `assignable this.*` (for constructors) or `assignable \nothing` (for all other specification cases).

4. Desugar specifications that are empty (i.e. have no specification cases) to the default specification (i.e. it makes minimal assumptions). If another method is overridden, then we add "`requires false;`" as to not change the meaning of the inherited specification. In all other cases, this usually results in simply adding "`requires true;`".

5. Desugar the use of nested specification cases in specifications from the inside out to eliminate nesting.

6. Desugar different behavioural cases to V `behaviour`, where V is the visibility level of the method or constructor being specified, mostly by filling the defaults for omitted clauses. This leaves only spec-cases that begin with V `behaviour`, for some visibility level V.

7. Combine each *extending-specification* (in a subclass or refinement) with the inherited or refined method specifications, mostly by copying inherited assertions from supertypes to subtypes, also including various kinds of renaming.

8. Desugar each *signals*-clause so that it refers to an exception of type `Exception` with a standard name by using a standard, but fresh, name for all exceptions (to allow combining all *signals*-clauses in the next step). In order to keep the originally declared exception's type, an `instanceof`-check is added.

9. Desugar the use of multiple clauses of the same kind, such as e.g. multiple *requires*-clauses. For *assignable*-clauses this consists in simply joining the lists, for *signals-only*-clauses the combination is the intersection (the exception type has to be a subtype of an exception type in each *signals-only*-clause) and for the other clauses (*requires*-clauses, *ensures*-clauses, *diverges*-clauses and *signals*-clauses) the process mostly simply conjoins the predicates in each clause. All these con- and disjunctions are done using the according short-circuit operator to allow clauses to be guarded by one another (depending on the operator used, cf. section 2.5).

10. Desugar each *signals-only*-clause by conjuncting the equivalent disjunction "$\bigvee_{E \in S}$ (*e* `instanceof` *E*)" (S denoting the set of all exception types E declared in the *signals-only*-clause and e denoting the exception variable) to the according *signals*-clause.

Whereas only *method specifications* are covered in this subsection, any specification element will be assumed to be fully *desugared* (in an a similar fashion as described by the steps above) throughout the text. In the following, a method specification will be assumed to boil down to the following specification elements:

```
/*@ V behaviour
  @  requires pre;
  @  measured_by measured;
  @  diverges div;
  @  accessible access;
  @  assignable assign;
  @  ensures post;
  @  signals (Exception exc) xpost;
  @*/
```

We call this a *method contract*, whereof several can be specified for one method (combined by the keyword `also`). The elements ***pre***, ***div***, ***post*** and ***xpost*** denote formulas, ***assign*** and ***access*** location sets, ***measured*** an integer, ***exc*** an exception variable and V a visibility. As they are not supported by KeY, the clauses `forall` (defining universal quantifications over the specification case), `old` (defining expressions to be evaluated in the pre-state), `when` (specifying concurrency aspects), `callable` (specifying which methods may be called), `captures` (specifying when an object, passed as an actual parameter, may be captured), `working_space` (specifying the maximum amount of heap space used) and `duration` (specifying the maximum processing time needed) are not covered in this work.

2.4. The KeY System

The KeY System[1] is a formal software development tool, which provides mechanisms for deductive proofs of the correctness of Java programs with respect to their JML* specification by using a theorem prover for the first-order Dynamic Logic for Java [BKS07]. In order to do this, a Java program including its specification gets translated into proof obligations. These are formulas whose validity corresponds to the correctness of the program with respect to its specifications. The underlying logic is JavaDL*, a dynamic logic for Java, which extends first order logic by modal operators holding executable Java program fragments. The subsequent verification is deductive and often automatic, where one can intervene to help interactively and occasionally must do so. A successful verification results in proving universal validity of the proof obligations. One concept significantly distinguishing JavaDL* from other dynamic logics is the concept of (state) *updates*. These are explicit syntactic representations of state changes, independent of the programming language. They allow a calculus to symbolically execute formulas in a way, which is equivalent to the actual program execution with symbolic values. Ultimately, symbolic execution removes all programs from the proof obligations, such that the verification process gets reduced to the problem of proving (universal) validity of formulas in first order logic. This task can be performed either directly by KeY or by another theorem prover. KeY is a joint project of Karlsruhe Institute of Technology and Chalmers University of Technology, Gothenburg and TU Darmstadt.

[1] `http://www.key-project.org/`

2.5. Well-Definedness Operator

As already indicated, there is a strong motivation to not only interpret JML boolean expressions in a standard *two-valued logic*, but to also consider a semantics, which is more programming-oriented and thus leaves room for exceptions similar to those in Java programs. Following the *rejection* approach (approach 6), we additionally introduce the *Well-Definedness Operators* $\mathcal{WD}$ (taking a formula as argument) and *wd* (taking a term as argument) as a semantic assertion for the specification's *well-definedness* (cf. [RDM08], [Bru09] and [Ulb09]). Their primary function is to tell whether Java would throw an exception on evaluation of the given specification expression and can be seen as a kind of second, orthogonal filter function. On the top-level, a boolean expression is taken to be valid **if and only if**[1] it evaluates to *true* and is *well-defined*. For most expressions, except the short-circuit versions of boolean operators, *wd* ($\mathcal{WD}$) is true **iff** *wd* ($\mathcal{WD}$) holds for every sub-expression.

It is important to note that these operators do **not** tell whether an expression is syntactically well-formed and subsequently syntactical correctness will be always assumed in this work. In particular, $\mathcal{WD}$ and *wd* do not report Java expressions, which result in an error at *compile time*, but exceptions which would occur at *run-time*. Although in general, Java expressions can have side-effects and thus $\mathcal{WD}$ and *wd* would need to depend on the state of evaluation, this can be neglected for most JML specifications, since it has either no effect on the specification's *well-definedness*, or it is not permitted in JML specifications, e.g. expressions as "`i++`" are not allowed in JML specifications. Exceptions are `\old` expressions and postconditions. For both, it is sufficient to distinguish two different states, the pre- and the post-state. In KeY this is done by distinguishing two different *heaps* and by performing *updates* (section 2.7.2) on these two. In the following, we will only distinguish between the two *Well-Definedness Operators* $\mathcal{WD}$ and *wd* from section 3.1.1 to section 3.1.9 and subsequently only denote *wd* for the appropriate choice for reasons of simplicity.

The semantics of these two operators should be seen more as a syntactical procedure than as a function in the sense that the argument is always interpreted as a whole and rules cannot be applied to sub-arguments. As such, it is similar to *predicate transformer semantics* as for example the well-known *weakest precondition* defined in [Dij75]. In the case of the KeY semantics, this means that the argument is seen as a purely syntactical construct, which is left uninterpreted until application of any of the *well-definedness* rules defined in the following chapter. This includes e.g. the rule of commutativity, and also other operations as e.g. *updates* (section 2.7.2) and instantiations are blocked until the argument is fully transformed by $\mathcal{WD}$ and *wd*, meaning that $\mathcal{WD}$ and *wd* do not occur anywhere in the scope of this operation (e.g. the *update*) anymore. In the further course of this work, we will denote this concept of syntactical transformation by the name *term transformer*.

In the following, we present three different implementations of $\mathcal{WD}$, as the difference only exists for boolean expressions. The reason for this limitation is simply the fact that in

[1] From now on, this will be abbreviated as **iff**.

Java, short-circuit operators only exist for boolean expressions. One might also think of applying both of these two different semantics for determining the *well-definedness* of e.g. a whole method contract, taking the precondition as the guarding and the postcondition as the guarded expression. This is, however, not a useful approach as JML models a method with a two-state semantics, where it is only executed – hence can only terminate and thus reach its post-state – **iff** its precondition holds in its pre-state. Consequently, an evaluation of the postcondition without evaluating the precondition contradicts the underlying blocking two-state semantics and is therefore no subject to consider for a useful *Well-Definedness Check*. Whereas the $\mathcal{D}$-operator and the $\mathcal{Y}$-operator are semantically equivalent and only differ technically, the $\mathcal{L}$-operator semantically differs from the former two. In the following chapters, we will only distinguish the $\mathcal{D}$-operator and the $\mathcal{L}$-operator, where the semantical difference matters. However, we will denote rules for all three operators in the next chapter.

2.5.1. The $\mathcal{D}$-Operator

This operator is complete and based on classical logic, where the order of terms and formulas is irrelevant. By complete, we imagine our semantics to be a three-valued one (i.e. formulas can result in either *true*, *false* or *undefined* ($\perp$)) and mean to not be left with any expression possibly resulting in the value *undefined*. As stated in the description for the approach of *rejection* (approach 6), this leaves us with a *pseudo-two-valued logic*, where the value *undefined* does not play any critical role. The $\mathcal{D}$-operator is based on strong Kleene logic. Direct implementations are rather inefficient, since formulas may blow up exponentially [BBM98]. In addition to the original definition, we give definitions for the ternary and `\ifEx` operators (section 3.1.9.2).

2.5.2. The $\mathcal{L}$-Operator

This operator is more intuitive for software developers and based on run-time assertion semantics. *Well-Definedness Checks* are stricter (i.e. all expressions which are *well-defined* under the $\mathcal{L}$-operator are automatically *well-defined* under the $\mathcal{D}$-operator, but not necessarily vice versa) using this operator, since the order of terms and formulas matters (using *short-circuit evaluation*). It is based on McCarthy logic and consequently stricter than the $\mathcal{D}$-operator. This also results in very efficient *Well-Definedness Checks* (linear with respect to the input formula) and according to our test results in section 5.2, it is the most efficient one of the three operators presented. There are surveys [Cha05] giving justification for choosing this comparatively strict semantics. One main reason presented there is the coping of user or practitioner needs, who accordingly do not want to adapt to different semantics, but rather support the use of one semantics for both programming language and program assertions. Since in Java the casting of exceptions is highly order-dependent (e.g. "`i != 0 && j/i >= 0`" differs from "`j/i >= 0 && i != 0`" exception-wise), this justifies JML to behave similarly.

2.5.3. The $\mathcal{Y}$-Operator

This operator is complete and based on classical logic, where the order of terms and formulas is irrelevant. It is semantically equivalent to the $\mathcal{D}$-operator, but more efficient as the conditions grow only linearly with respect to the input formula, since it postpones the distinction of cases to the end of the evaluation [DMR08]. The process of postponing is done using the two operators $\mathcal{T}$ and $\mathcal{F}$, meaning the argument to be *well-defined* and evaluate to *true* or *false* respectively. Thus, the *Well-Definedness Check* consists in evaluating the disjunction of these two operators.

2.6. Black Box Semantics

Throughout this whole work, we consider JML specifications and JML statements independently of their enclosed, but very well dependent of the JML specification and Java code, in which they are embedded (i.e. their *context*). For example, we assume a method specification to know about the class invariant of the class in which it is embedded, but not about any code or specifications **inside** the method it specifies (it does know about the method's signature though). Similarly, a JML loop statement is assumed to know about the precondition of the method in which it is embedded and also the method's Java code being executed prior to the loop. However, it does not know about anything happening **inside** the actual loop (it can talk about the loop's guard though).
A justification for this semantics or assumption is, that the *well-definedness* of any JML specification should not depend on the code being specified, but be seen solely on specification level.

2.7. State Semantics

As in JML specifications, especially in method contracts, we distinguish between two different states, namely the pre- and the post-state, this also has to be considered when deciding about their *well-definedness*. This being said, our black box assumption prevents us from knowing the exact method code or what happens between those two states.
Nevertheless, expressions in a JML method contract can very well be *well-defined* in the precondition but not in the postcondition and vice versa, especially when talking about references. This is due to changes made by the code, e.g. when a global reference is set to `null`, it can subsequently lead to a `NullPointerException`. Only looking at the JML specification, the sole information about state changes we have lies in the *assignable*-clause, which specifies the locations potentially to be changed in the method. In KeY, the modelling of locations visible in method specifications is done by the usage of an explicit *heap* and performing changes, e.g. incrementing the value of an integer variable, on it through *updates*.

2.7.1. The Heap

The *heap* can be seen as representation of a global system state when no execution is in progress. Practically, it is a mapping of all locations persistent throughout program execution to their domains [Ulb08]. For our matter, the *assignable*-clause specifies a subset of those locations. KeY lets us use *updates* to express those changes on the *heap*. The only problem left is that we do not know about the exact nature of those changes and only know, **which** locations can be changed, but not **how**. We would like to have a mechanism to describe the set $H_L(s)$ of "havoc"ed states having these possibly changed locations (here denoted by the set L) set to arbitrary values, but coinciding on the rest of the original *heap* with the state s [Ulb09]. For this matter, KeY gives us the feature of *heap anonymisation* using an *anonymous heap*.

2.7.2. Updates

Updates enhance the JavaDL (Dynamic Logic for Java) calculus with the ability to handle Java programs by doing a *forward symbolic execution* of the program. This is done by stepwise turning all program assignments into *updates* to represent the state changes. For e.g. u being an update and *post* a JavaDL formula, "$\{u\}post$" is also a JavaDL formula. If, for example, we apply the *update* in the formula "`{i := 3} i > 0`", we get "`3 > 0`", which results to *true*. In general, there are sequential and parallel *updates*, differing in the order in which they are applied.

2.7.3. The Stack

But as there are more elements in JML, for example loop statements (section 3.3.1) and block contracts (section 3.3.2) which can be found in the code and thus talk about possibly changed (compared to the pre-state of the embedding method) variables and references not persistent throughout program execution, we also need to consider the *stack*.
The *stack* stores local variables and method parameters, local meaning local to the current method frame. Changes to the *stack* can also be expressed via *updates*, but now those changes are explicit and not arbitrary. The reason for this apparent deviation from our black box assumption is that any code for a loop, block or similar is always seen and considered in *context*, otherwise it would be written in its own method. Consequently it would not make sense for the JML statement (e.g. a loop statement including a loop invariant) to semantically differ from the annotated code (e.g. a loop). KeY also verifies loop statements and block contracts as part of their embedding method and not as separate or independent elements.

3. Formal Definitions

Generally formal specifications are *well-defined* **iff** all their partial operations are applied within their domain. Then these specifications do not depend on ambiguous, i.e. ill- or undefined, terms and always evaluate to either *true* or *false*, regardless of their ill- or undefined subterms' interpretations. E.g. a method call would hence be *well-defined* **iff** its receiver is not equal to `null` and the called method's precondition is satisfied. In this chapter, we give *well-definedness* definitions for a major part of the expressions and specification elements in JML*, strongly focussing on the ones supported by KeY. However, these definitions will be given for their arguments being formulas or expressions in JavaDL as KeY does some effort on pretranslating the given specification, e.g. by adding certain assumptions as for example for non-static non-constructor methods that "`this`" is an exact instance of the method's class. Both the argument and the result of the *Well-Definedness Operator* are expressions of JavaDL*. Nonetheless, we will mostly stick to the better-known JML/Java syntax whenever possible, also to state that a syntactical expression is taken and a semantical one returned.

3.1. Predicates and Specification Expressions

In this section, we give definitions for the *well-definedness* of predicates and specification expressions, forming the lower-level elements in JML/JML*. Many of them are equal or similar to a Java expression or predicate, as JML/JML*'s expression language is almost a superset of Java expressions. They also include location sets, which are used to describe locations instead of values.

3.1.1. Primary Terms and Formulas

As *wd* and $\mathcal{WD}$ operate purely syntactically, expressions with no sub-arguments are trivially *well-defined*. In the following, let f be a constant predicate symbol and t a constant function symbol.

$$\mathcal{WD}(f) \triangleq \mathit{true}$$
$$\mathit{wd}(t) \triangleq \mathit{true}$$

3.1.2. General Expressions and Predicates

For the *well-definedness* of typecasts, the `instanceof`-condition is added in order to detect a possible throw of a `ClassCastException`. The *exactInstance* expression, denoting a

term to be exactly of a specific type (as opposed to the usual `instanceof`-expression, this does not hold for subtypes), is not part of JML but JavaDL, and added in the pretranslation process done by KeY.
In the following, let s and t be terms of compatible types and T a type.

$$\begin{aligned} wd((T)t) &\triangleq wd(t) \land (t \texttt{ instanceof } T) \\ wd(exactInstance_T(t)) &\triangleq wd(t) \\ wd(t \texttt{ instanceof } T) &\triangleq wd(t) \\ \mathcal{WD}(s \texttt{ == } t) &\triangleq wd(s) \land wd(t) \\ \mathcal{WD}(s \texttt{ != } t) &\triangleq wd(s) \land wd(t) \end{aligned}$$

3.1.3. Numerical Expressions

Most of the numerical expressions in JML* are standard JML. However, in JML* the quantifiers `\sum` and `\prod` are replaced by `\bsum` and `\bprod`, which are their bounded equivalents. An expression "`(\bsum` T i`;` a`;` b`;` c`)`" is thus the equivalent to "`(\sum` T i`;` a `<=` i `&&` i `<` b`;` c`)`" and likewise for `\prod` and `\bprod`. The quantifier `\bsum` can also be used to specify the numerical quantifier `\num_of`, similar to the description in [Lea+11]. For the *well-definedness* of numerical expressions, the main concern is to handle the possible cast of an `ArithmeticException`, e.g. in case of a division by zero. In expressions using the quantifiers `\bsum` or `\bprod`, the *well-definedness* of the last argument only propagates in case it actually is evaluated. If there is no case of actual evaluation of this argument, its *well-definedness* does not propagate at all as it does not influence the end result. However, we do not handle the possible occurrence of any numerical overflow as this is considered to be a valid operation in the semantics of programming languages as e.g. Java or C++.
In the following, let T be a type, i an identifier, a, b and c expressions of exact numerical types, $\circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{>>}, \texttt{<<}, \texttt{>>>}, \texttt{|}, \texttt{\&}, \texttt{\^{}}\}$ and $\diamond \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$. Also i does not appear freely in a or b.

$$\begin{aligned} wd(\texttt{-}a) &\triangleq wd(a) \\ wd(\texttt{\~{}}a) &\triangleq wd(a) \\ wd(a \circ b) &\triangleq wd(a) \land wd(b) \\ wd(a \texttt{ / } b) &\triangleq wd(a) \land wd(b) \land (b \neq 0) \\ wd(a \texttt{ \% } b) &\triangleq wd(a) \land wd(b) \land (b \neq 0) \\ wd(\texttt{\textbackslash bsum } T\ i\texttt{;}\ a\texttt{;}\ b\texttt{;}\ c) &\triangleq wd(a) \land wd(b) \\ &\quad \land \forall\, T\ i\colon (a \leq i \land i < b) \rightarrow wd(c) \\ wd(\texttt{\textbackslash bprod } T\ i\texttt{;}\ a\texttt{;}\ b\texttt{;}\ c) &\triangleq wd(a) \land wd(b) \\ &\quad \land \forall\, T\ i\colon (a \leq i \land i < b) \rightarrow wd(c) \\ \mathcal{WD}(a \diamond b) &\triangleq wd(a) \land wd(b) \end{aligned}$$

3.1.4. Location Sets

As KeY uses dynamic frames, location sets (i.e. sets of memory locations) are elevated to *first-class citizens* of JML* and specification expressions are enabled to talk about them directly. This includes statements about not overlapping (`\disjoint`), that some concrete location or location set is or is not part of a particular location set (*elementOf*, `\subset`), to form an infinite union in the mathematical sense (`\infinite_union`) and other standard set theoretical operators. The predicate *createdInHeap* can result from the pretranslation process in KeY and states that some location set l is already created in a specific heap h.
For the *well-definedness* of location sets, the exception types `NullPointerException`, `NegativeArraySizeException` and `IndexOutOfBoundsException`, in particular the `ArrayIndexOutOfBoundsException`, are handled. For fields, we need to distinguish between static and non-static ones, as in the static case checking the referenced object for an equality to `null` does not make sense. As in KeY static references are distinguished from non-static ones by the usage of `null` instead of an actual object, the check has to be done in this manner.
In the following, let T be a type, o an object, a an array object, b a non-array object, h a heap, i an identifier, f a field, s a static field, t a non-static field, k and l location sets and m and n expressions of exact numerical types.

$$
\begin{aligned}
wd(a\texttt{[*]}) &\triangleq wd(a) \land (a \neq \texttt{null}) \\
wd(b.\texttt{*}) &\triangleq wd(b) \\
wd(\texttt{\textbackslash all\_objects}(f)) &\triangleq wd(f) \\
wd(b.t) &\triangleq wd(b) \land wd(t) \land (b \neq \texttt{null}) \\
wd(b.s) &\triangleq wd(b) \land wd(s) \\
wd(a\texttt{[}n\texttt{]}) &\triangleq wd(a) \land wd(n) \land (a \neq \texttt{null}) \\
&\quad \land 0 \leq n \land n < a.\texttt{length} \\
wd(\texttt{\textbackslash set\_union}(k,l)) &\triangleq wd(k) \land wd(l) \\
wd(\texttt{\textbackslash intersect}(k,l)) &\triangleq wd(k) \land wd(l) \\
wd(\texttt{\textbackslash set\_minus}(k,l)) &\triangleq wd(k) \land wd(l) \\
wd(a\texttt{[}m\texttt{..}n\texttt{]}) &\triangleq wd(a) \land wd(m) \land wd(n) \land (a \neq \texttt{null}) \\
&\quad \land 0 \leq m \land m \leq n \land n < a.\texttt{length} \\
wd(\texttt{\textbackslash infinite\_union } T\ i\texttt{;}\ l) &\triangleq \forall\, T\ i\colon wd(l) \\
\mathcal{WD}(elementOf(o.t,l)) &\triangleq wd(o) \land wd(t) \land wd(l) \land (o \neq \texttt{null}) \\
\mathcal{WD}(elementOf(o.s,l)) &\triangleq wd(o) \land wd(s) \land wd(l) \\
\mathcal{WD}(\texttt{\textbackslash subset}(k,l)) &\triangleq wd(k) \land wd(l) \\
\mathcal{WD}(\texttt{\textbackslash disjoint}(k,l)) &\triangleq wd(k) \land wd(l)
\end{aligned}
$$

$$\mathcal{WD}(\mathit{createdInHeap}(l,h)) \triangleq \mathit{wd}(l) \land \mathit{wd}(h) \land \mathit{wellFormed}(h)$$
$$\mathcal{WD}(\texttt{\textbackslash fresh}(h,l)) \triangleq \mathit{wd}(l) \land \mathit{wd}(h) \land \mathit{wellFormed}(h)$$

3.1.5. Heap Expressions

As KeY uses an explicit heap model, the changes on it also need to be represented. As such, we talk about object creations (*create*), a specific symbol denoting created objects (*o.created* for some created object *o*), storing operations (*store*) similar to the storing operation in array theory, heap anonymisation (*anon*) (section 2.7.1) and *well-formedness* of heaps (*wellFormed*). The predicate *wellFormed* states for a heap h that all objects stored in h are either the `null` object or are created in h, that all location sets stored in h contain only locations belonging to objects that are created or to `null` and finally that only finitely many objects may be created in h. This is again a predicate resulting from the pretranslation process in KeY as these are properties assumed for any fresh heap before a program starts. It is a property maintained by any terminating Java program. The syntax $h[e]$ denotes a read access of the element e on a heap h.
What is handled by these definitions is less an actual cast of an exception than it is to make sure that we only make statements about actually created objects in the state of the expression. This cannot occur in actual Java programs, but very well in JML/JML* specifications, and as such, we will refer to it by the name `ObjectNotCreatedException`. For example, the postcondition "`\old(o) != null`" should *not* be *well-defined* for an object o which did not exist in the pre-state.
In the following, let o be an object, a an array object, b a non-array object, g and h heaps, f a field, r a non-static field, s a static field, l a location set, n an expression of exact numerical type and t a term.

$$\begin{aligned}
\mathit{wd}(a.\texttt{length}) &\triangleq \mathit{wd}(a) \land (a \neq \texttt{null}) \\
\mathit{wd}(\mathit{create}(h,o)) &\triangleq \mathit{wd}(h) \land \mathit{wd}(o) \land \mathit{wellFormed}(h) \land (o \neq \texttt{null}) \\
\mathit{wd}(h[o.\mathit{created}]) &\triangleq \mathit{wd}(h) \land \mathit{wd}(o) \land \mathit{wellFormed}(h) \land (o \neq \texttt{null}) \\
\mathit{wd}(h[b.r]) &\triangleq \mathit{wd}(h) \land \mathit{wd}(b) \land \mathit{wd}(r) \\
&\quad \land \mathit{wellFormed}(h) \land (b \neq \texttt{null}) \\
&\quad \land ((r = \mathit{created}) \lor h[b.\mathit{created}]) \\
\mathit{wd}(h[b.s]) &\triangleq \mathit{wd}(h) \land \mathit{wd}(b) \land \mathit{wd}(s) \\
&\quad \land \mathit{wellFormed}(h) \\
\mathit{wd}(h[a\texttt{[}n\texttt{]}]) &\triangleq \mathit{wd}(h) \land \mathit{wd}(a) \land \mathit{wd}(n) \land \mathit{wellFormed}(h) \\
&\quad \land (a \neq \texttt{null}) \land h[a.\mathit{created}] \\
&\quad \land 0 \leq n \land n < a.\texttt{length} \\
\mathit{wd}(\mathit{anon}(h,l,g)) &\triangleq \mathit{wd}(h) \land \mathit{wd}(l) \land \mathit{wd}(g) \land \mathit{wellFormed}(h) \\
&\quad \land \mathit{wellFormed}(g)
\end{aligned}$$

$$\begin{aligned} wd(store(h, o.f, t)) &\triangleq wd(h) \wedge wd(o) \wedge wd(f) \wedge wd(t) \wedge wellFormed(h) \\ &\quad \wedge (o \neq \texttt{null}) \wedge h[o.created] \\ \mathcal{WD}(wellFormed(h)) &\triangleq wd(h) \\ \mathcal{WD}(\texttt{\textbackslash fresh}(h, t)) &\triangleq wd(l) \wedge wd(h) \wedge wellFormed(h) \end{aligned}$$

3.1.6. Reach Predicates

The `\reach`-predicate in JML* slightly differs from its counterpart in JML [Lea+11]. In the following, let h be a heap, l a location set, o and p objects and n an expression of exact numerical type. In JML*, it is of type boolean and expresses that from object o, object p can be reached through a location in l in exactly n steps. The `\acc`-predicate expresses that p is a field in object o belonging to the location set l. Here, we basically only propagate the *well-definedness*. Non-null checks for the objects o and p or any conditions for potentially ambiguous values of n are not needed as in these cases, the predicates evaluate unambiguously to *false*.

$$\begin{aligned} \mathcal{WD}(\texttt{\textbackslash acc}(h, l, o, p)) &\triangleq wd(h) \wedge wd(l) \wedge wd(o) \wedge wd(p) \wedge wellFormed(h) \\ \mathcal{WD}(\texttt{\textbackslash reach}(h, l, o, p, n)) &\triangleq wd(h) \wedge wd(l) \wedge wd(o) \wedge wd(p) \wedge wd(n) \wedge wellFormed(h) \end{aligned}$$

3.1.7. Sequences

Another extension to original JML as defined in [Lea+11] is the sequence data type, representing finite mathematical sequences. This data type is formulated in a natural way by using basic mathematical concepts such as sets and relations, and subsequently facilitates the specification process in KeY.
Its elementary constructor is `\seq_singleton` taking an object as argument. However, a sequence can also be instantiated by using the quantified constructor `\seq_def`. The function s.`length` denotes the length of a sequence s, `\seq_reverse` reverses the order of a given sequence, `\seq_n_perm_inv` inverts a given permutation, `s[n]` returns the element at a given position n in the sequence s, `\seq_index_of` returns the (first) index for an element in the sequence, `\seq_concat` concatenates two sequences, `\seq_remove` removes the element at a given position in the sequence, s`[`$m..n$`]` denotes the subsequence of s in the given boundaries, `\seq_swap` swaps two elements. The predicate `\seq_perm` tells whether one sequence is a permutation of another and `\seq_n_perm` states a sequence to be a permutation.
Although sequences cannot be part of any Java expression, we imagine them to throw exceptions similarly to arrays or linked lists. The following rules handle the virtual cast of exceptions as `NegativeArraySizeException` and `IndexOutOfBoundsException`, in particular the `ArrayIndexOutOfBoundsException` if the sequence represents an array. Let below r and s be sequences, m and n expressions of exact numerical types, T a type, i an identifier and t a term. Also i does not appear freely in m and n.

$$
\begin{aligned}
wd(s.\texttt{length}) &\triangleq wd(s)\\
wd(\texttt{\textbackslash seq\_singleton}(t)) &\triangleq wd(t)\\
wd(\texttt{\textbackslash seq\_reverse}(s)) &\triangleq wd(s)\\
wd(\texttt{\textbackslash seq\_n\_perm\_inv}(s)) &\triangleq wd(s)\\
wd(s\texttt{[n]}) &\triangleq wd(s) \land wd(n) \land 0 \leq n \land n < s.\texttt{length}\\
wd(\texttt{\textbackslash seq\_index\_of}(s,t)) &\triangleq wd(s) \land wd(t)\\
wd(\texttt{\textbackslash seq\_concat}(r,s)) &\triangleq wd(r) \land wd(s)\\
wd(\texttt{\textbackslash seq\_remove}(s,n)) &\triangleq wd(s) \land wd(n) \land 0 \leq n \land n < s.\texttt{length}\\
wd(s\texttt{[}m\texttt{..}n\texttt{]}) &\triangleq wd(s) \land wd(m) \land wd(n) \land 0 \leq m\\
&\quad \land m \leq n \land n \leq s.\texttt{length}\\
wd(\texttt{\textbackslash seq\_swap}(s,m,n)) &\triangleq wd(s) \land wd(m) \land wd(n) \land 0 \leq m \land 0 \leq n\\
&\quad \land m < s.\texttt{length} \land n < s.\texttt{length}\\
wd(\texttt{\textbackslash seq\_def(}T\ i\texttt{;}\ m\texttt{;}\ n\texttt{;}\ t\texttt{)}) &\triangleq wd(m) \land wd(n) \land m \leq n\\
&\quad \land \forall\ T\ i\colon (m \leq i \land i < n) \rightarrow wd(t)\\
\mathcal{WD}(\texttt{\textbackslash seq\_perm}(r,s)) &\triangleq wd(r) \land wd(s)\\
\mathcal{WD}(\texttt{\textbackslash seq\_n\_perm}(s)) &\triangleq wd(s)
\end{aligned}
$$

3.1.8. String Expressions

Although we named and titled them to be strings, in JML* they are actually represented as lists of characters and characters can also be represented numerically. Most of the following functions denote the intuitively named operations and do as such not need much further explanation. The *undefinedness* problems handled by the following rules are casts of exceptions as `NegativeArraySizeException` and `IndexOutOfBoundsException`.
In the following, let s and t be strings and m and n expressions of exact numerical types.

$$
\begin{aligned}
wd(s.\texttt{length}) &\triangleq wd(s)\\
wd(\texttt{\textbackslash translate\_int}(n)) &\triangleq wd(n)\\
wd(\texttt{\textbackslash remove\_zeros}(s)) &\triangleq wd(s)\\
wd(\texttt{\textbackslash hash\_code}(s)) &\triangleq wd(s)\\
wd(\texttt{\textbackslash cat}(s,t)) &\triangleq wd(s) \land wd(t)\\
wd(\texttt{\textbackslash cons}(n,s)) &\triangleq wd(n) \land wd(s) \land 0 \leq n\\
wd(\texttt{\textbackslash char\_at}(n,s)) &\triangleq wd(n) \land wd(s) \land 0 \leq n \land n < s.\texttt{length}\\
wd(\texttt{\textbackslash concat}(s,t)) &\triangleq wd(s) \land wd(t)
\end{aligned}
$$

$$
\begin{aligned}
wd(\texttt{\textbackslash index\_of\_char}(m,n,s)) &\triangleq wd(m) \wedge wd(n) \wedge wd(s) \\
&\quad \wedge 0 \leq m \wedge m \leq n \wedge n < s.\texttt{length} \\
wd(\texttt{\textbackslash sub}(m,n,s)) &\triangleq wd(m) \wedge wd(n) \wedge wd(s) \\
&\quad \wedge 0 \leq m \wedge m < n \wedge n < s.\texttt{length} \\
wd(\texttt{\textbackslash index\_of\_cl}(s,n,t)) &\triangleq wd(s) \wedge wd(n) \wedge wd(t) \\
&\quad \wedge 0 \leq n \wedge n + s.\texttt{length} < t.\texttt{length} \\
wd(\texttt{\textbackslash last\_index\_of\_char}(m,n,s)) &\triangleq wd(m) \wedge wd(n) \wedge wd(s) \\
&\quad \wedge 0 \leq m \wedge m \leq n \wedge n < s.\texttt{length} \\
wd(\texttt{\textbackslash last\_index\_of\_cl}(s,n,t)) &\triangleq wd(s) \wedge wd(n) \wedge wd(t) \\
&\quad \wedge 0 \leq n \wedge n + s.\texttt{length} < t.\texttt{length} \\
wd(\texttt{\textbackslash replace}(m,n,s)) &\triangleq wd(m) \wedge wd(n) \wedge wd(s) \wedge 0 \leq m \wedge 0 \leq n \\
&\quad \wedge m < s.\texttt{length} \ \wedge n < s.\texttt{length} \\
\mathcal{WD}(\texttt{\textbackslash starts\_with}(s,t)) &\triangleq wd(s) \wedge wd(t) \\
\mathcal{WD}(\texttt{\textbackslash ends\_with}(s,t)) &\triangleq wd(s) \wedge wd(t) \\
\mathcal{WD}(\texttt{\textbackslash contains}(s,t)) &\triangleq wd(s) \wedge wd(t)
\end{aligned}
$$

3.1.9. Boolean Expressions

In the following, we present *well-definedness* rules for the three different operators $\mathcal{L}$, $\mathcal{D}$ and $\mathcal{Y}$. As already mentioned in section 2.5, the former two are semantically equivalent but their implementations highly differ in performance. Regarding their notation, we will use the operator symbol $\mathcal{WD}$ or wd – marked by an according subscript however – for all three[1] of them as they do not interact on each other, but each stands separately as an alternative to the other two, possibly having a different result if they differ semantically. Most of the covered boolean operators also exist in JML, except for the `\ifEx` operator, which is specific to JML*. On the one hand, it is used to specify statements using the quantifiers `\max` or `\min` and on the other hand, it allows us to make even more expansive statements. As for the generalised quantifiers `\min` and `\max`, an expression of the form "(`\max` T i; a; n)" translates to "`\ifEx` T i; $(a \wedge \forall\ T\ i';\ a' \rightarrow n \leq i')$ `\then`(n) `\else`(u)", wherein T is a type, i an identifier, a an expression of type boolean, n an expression of numerical type and u a constant numerical function denoting *undefMin*, which stands for an undefined value similar to $\perp$. The primed expressions denote their unprimed counterparts, wherein i is substituted to i'. For the quantifier `\max`, the same applies, except that "$\leq$" becomes "$\geq$".
As for more expansive statements and the general semantics of the `\ifEx` operator, let i be an identifier, T a type, φ a boolean function, and ν a function of compatible type to the constant function u. Furthermore, i does not appear freely in u. Then an

[1] However, the $\mathcal{Y}$-operator additionally uses two further operators $\mathcal{T}$ and $\mathcal{F}$, which only act on an interim basis though.

expression of the form "\ifEx T i; $\varphi(i)$ \then$(\nu(i))$ \else$(u)$" can be axiomatised by the following statement, where the binary relation "$x \preceq y$" denotes the total well-order relation "$(y \leq x \wedge y < 0) \vee (0 \leq x \wedge x \leq y)$":

$$\Big((\exists x\colon \varphi(x)) \rightarrow \Big(\forall x\colon \varphi(x) \wedge (\forall y\colon \varphi(y) \rightarrow x \preceq y) \rightarrow \Big[(\texttt{\ifEx } T \ i\texttt{; } \varphi(i) \ \texttt{\then}(\nu(i)) \ \texttt{\else}(u)) = \nu(x)\Big]\Big)\Big)$$
$$\wedge \Big((\forall x\colon \neg\varphi(x)) \rightarrow \Big[(\texttt{\ifEx } T \ i\texttt{; } \varphi(i)\,\texttt{\then}(\nu(i))\,\texttt{\else}(u)) = u\Big]\Big).$$

In the following rules, we mostly only propagate *well-definedness*, but depending on which operator we choose, this propagation is done in a different way and hence may interpret specifications to be *undefined* in the $\mathcal{L}$-operator semantics, which would be *well-defined* in the $\mathcal{D}$- and $\mathcal{Y}$-operator semantics. The exception is the \ifEx operator, for which we additionally need to check that there is only one minimal x, for which the guard holds.

3.1.9.1. $\mathcal{L}$-Operator

This operator represents a semantics similar to the expression semantics of most modern programming languages such as Java and is the strictest of them all as it also considers the order of expressions where short-circuit operators are (implicitly or explicitly) used. In the following, let a, b and c be expressions of type boolean and d and e terms of compatible types.

$$\begin{aligned}
\mathcal{WD}_{\mathcal{L}}(\texttt{!}\,a) &\triangleq \mathcal{WD}_{\mathcal{L}}(a)\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{\&}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge \mathcal{WD}_{\mathcal{L}}(b)\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{\&\&}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge (a \rightarrow \mathcal{WD}_{\mathcal{L}}(b))\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{|}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge \mathcal{WD}_{\mathcal{L}}(b)\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{||}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge (\neg a \rightarrow \mathcal{WD}_{\mathcal{L}}(b))\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{==>}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge (a \rightarrow \mathcal{WD}_{\mathcal{L}}(b))\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{<==}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(b) \wedge (b \rightarrow \mathcal{WD}_{\mathcal{L}}(a))\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{<==>}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge \mathcal{WD}_{\mathcal{L}}(b)\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{<=!=>}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge \mathcal{WD}_{\mathcal{L}}(b)\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{\^{}}\,b) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge \mathcal{WD}_{\mathcal{L}}(b)\\
wd_{\mathcal{L}}(a\,\texttt{?}\,d : e) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge (a \rightarrow wd_{\mathcal{L}}(d)) \wedge (\neg a \rightarrow wd_{\mathcal{L}}(e))\\
\mathcal{WD}_{\mathcal{L}}(a\,\texttt{?}\,b : c) &\triangleq \mathcal{WD}_{\mathcal{L}}(a) \wedge (a \rightarrow \mathcal{WD}_{\mathcal{L}}(b)) \wedge (\neg a \rightarrow \mathcal{WD}_{\mathcal{L}}(c))
\end{aligned}$$

Let furthermore below T be a type and i an identifier, not appearing freely in c and e.

$$\begin{aligned}
wd_{\mathcal{L}}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(d)\,\texttt{\textbackslash else}(e)) &\triangleq (\forall\ T\ i\colon \mathcal{WD}_{\mathcal{L}}(a)) \\
&\quad \wedge (\forall\ T\ i\colon (a \to wd_{\mathcal{L}}(d))) \\
&\quad \wedge ((\forall\ T\ i\colon \neg a) \to wd_{\mathcal{L}}(e)) \\
\mathcal{WD}_{\mathcal{L}}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(b)\,\texttt{\textbackslash else}(c)) &\triangleq (\forall\ T\ i\colon \mathcal{WD}_{\mathcal{L}}(a)) \\
&\quad \wedge (\forall\ T\ i\colon (a \to \mathcal{WD}_{\mathcal{L}}(b))) \\
&\quad \wedge ((\forall\ T\ i\colon \neg a) \to \mathcal{WD}_{\mathcal{L}}(c)) \\
\mathcal{WD}_{\mathcal{L}}(\texttt{\textbackslash forall}\ T\ i\texttt{;}\ a\texttt{;}\ b) &\triangleq \forall\ T\ i\colon \mathcal{WD}_{\mathcal{L}}(a \texttt{ ==> } b) \\
\mathcal{WD}_{\mathcal{L}}(\texttt{\textbackslash exists}\ T\ i\texttt{;}\ a\texttt{;}\ b) &\triangleq \forall\ T\ i\colon \mathcal{WD}_{\mathcal{L}}(a \texttt{ \&\& } b)
\end{aligned}$$

3.1.9.2. $\mathcal{D}$-Operator

This operator represents a semantics based on Kleene logic in a more mathematical sense than the $\mathcal{L}$-operator and is hence less strict (i.e. the expressions which are *well-defined* under the $\mathcal{D}$-operator are a superset of the ones *well-defined* under the $\mathcal{L}$-operator) as the order of the expressions does not matter. Additionally to the original definitions given in [BBM98], we formulated rules for the ternary and the `\ifEx` operator. In the following, let a, b and c be expressions of type boolean and d and e terms of compatible types.

$$\begin{aligned}
\mathcal{WD}_{\mathcal{D}}(\texttt{!}a) &\triangleq \mathcal{WD}_{\mathcal{D}}(a) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ \& } b) &\triangleq (\mathcal{WD}_{\mathcal{D}}(a) \wedge \neg a) \vee (\mathcal{WD}_{\mathcal{D}}(b) \wedge \neg b) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b)) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ \&\& } b) &\triangleq (\mathcal{WD}_{\mathcal{D}}(a) \wedge \neg a) \vee (\mathcal{WD}_{\mathcal{D}}(b) \wedge \neg b) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b)) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ | } b) &\triangleq (\mathcal{WD}_{\mathcal{D}}(a) \wedge a) \vee (\mathcal{WD}_{\mathcal{D}}(b) \wedge b) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b)) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ || } b) &\triangleq (\mathcal{WD}_{\mathcal{D}}(a) \wedge a) \vee (\mathcal{WD}_{\mathcal{D}}(b) \wedge b) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b)) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ ==> } b) &\triangleq (\mathcal{WD}_{\mathcal{D}}(a) \wedge \neg a) \vee (\mathcal{WD}_{\mathcal{D}}(b) \wedge b) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b)) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ <== } b) &\triangleq (\mathcal{WD}_{\mathcal{D}}(b) \wedge \neg b) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge a) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b)) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ <==> } b) &\triangleq \mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ <=!=> } b) &\triangleq \mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ ^ } b) &\triangleq \mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b) \\
wd_{\mathcal{D}}(a \texttt{ ? } d : e) &\triangleq (\mathcal{WD}_{\mathcal{D}}(a) \wedge wd_{\mathcal{D}}(d) \wedge a) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge wd_{\mathcal{D}}(e) \wedge \neg a) \\
&\quad \vee (wd_{\mathcal{D}}(d) \wedge wd_{\mathcal{D}}(e) \wedge (d = e)) \\
\mathcal{WD}_{\mathcal{D}}(a \texttt{ ? } b : c) &\triangleq (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b) \wedge a) \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(c) \wedge \neg a) \\
&\quad \vee (\mathcal{WD}_{\mathcal{D}}(b) \wedge \mathcal{WD}_{\mathcal{D}}(c) \wedge (b \leftrightarrow c))
\end{aligned}$$

Let furthermore below T be a type, i an identifier, which does not appear freely in c and e, i' a fresh identifier, in which i does not appear freely, and a' the counterpart to a,

wherein i is substituted to i'.

$$
\begin{aligned}
wd_{\mathcal{D}}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(d)\,\texttt{\textbackslash else}(e)) \triangleq\ & \Big(\exists\, T\ i\colon \mathcal{WD}_{\mathcal{D}}(a) \wedge wd_{\mathcal{D}}(d) \wedge a \\
& \quad \wedge \forall\, T\ i'\colon \Big((i' \prec i) \rightarrow (\mathcal{WD}_{\mathcal{D}}(a') \wedge \neg a')\Big)\Big) \\
& \vee \Big(\forall\, T\ i\colon (\mathcal{WD}_{\mathcal{D}}(a) \wedge wd_{\mathcal{D}}(e) \wedge \neg a)\Big) \\
& \vee \Big(\forall\, T\ i\colon (wd_{\mathcal{D}}(d) \wedge wd_{\mathcal{D}}(e) \wedge (d = e))\Big) \\
\mathcal{WD}_{\mathcal{D}}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(b)\,\texttt{\textbackslash else}(c)) \triangleq\ & \Big(\exists\, T\ i\colon \mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b) \wedge a \\
& \quad \wedge \forall\, T\ i'\colon \Big((i' \prec i) \rightarrow (\mathcal{WD}_{\mathcal{D}}(a') \wedge \neg a')\Big)\Big) \\
& \vee \Big(\forall\, T\ i\colon (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(c) \wedge \neg a)\Big) \\
& \vee \Big(\forall\, T\ i\colon (\mathcal{WD}_{\mathcal{D}}(b) \wedge \mathcal{WD}_{\mathcal{D}}(c) \wedge (b \leftrightarrow c))\Big) \\
\mathcal{WD}_{\mathcal{D}}(\texttt{\textbackslash forall}\ T\ i\texttt{;}\ a\texttt{;}\ b) \triangleq\ & (\exists\, T\ i\colon (\mathcal{WD}_{\mathcal{D}}(a\,\texttt{==>}\,b) \wedge a \wedge \neg b)) \\
& \vee (\forall\, T\ i\colon \mathcal{WD}_{\mathcal{D}}(a\,\texttt{==>}\,b)) \\
\mathcal{WD}_{\mathcal{D}}(\texttt{\textbackslash exists}\ T\ i\texttt{;}\ a\texttt{;}\ b) \triangleq\ & (\exists\, T\ i\colon (\mathcal{WD}_{\mathcal{D}}(a\,\texttt{\&\&}\,b) \wedge (\neg a \vee b))) \\
& \vee (\forall\, T\ i\colon ;\mathcal{WD}_{\mathcal{D}}(a\,\texttt{\&\&}\,b))
\end{aligned}
$$

3.1.9.3. $\mathcal{Y}$-Operator

This operator represents the same semantics as the $\mathcal{D}$-operator, based on Kleene logic, but is defined in a much more efficient way (the size of the *well-definedness* conditions grows linearly with respect to the input formula), because it postpones the distinction of cases to the end of the evaluation. In addition to the original definitions given in [DMR08], we formulated rules for the ternary and the `\ifEx` operator based on the according rules formulated for the $\mathcal{D}$-operator.
In the following, let f be a predicate symbol with further formulas as arguments, g a predicate symbol with no further formulas as arguments (i.e. either no arguments or only terms as arguments) and $\mathcal{T}$ and $\mathcal{F}$ *term transformers* taking a formula as argument, operating similarly to $\mathcal{WD}$.

$$
\begin{aligned}
\mathcal{WD}_{\mathcal{Y}}(f) &\triangleq \mathcal{T}(f) \vee \mathcal{F}(f) \\
\mathcal{T}(g) &\triangleq \mathcal{WD}_{\mathcal{Y}}(g) \wedge g \\
\mathcal{F}(g) &\triangleq \mathcal{WD}_{\mathcal{Y}}(g) \wedge \neg g
\end{aligned}
$$

Let below a, b and c be boolean expressions and d and e terms of compatible types.

$$
\begin{aligned}
\mathcal{T}(\,!\,a) &\triangleq \mathcal{F}(a)\\
\mathcal{F}(\,!\,a) &\triangleq \mathcal{T}(a)\\
\mathcal{T}(a\,\&\,b) &\triangleq \mathcal{T}(a) \wedge \mathcal{T}(b)\\
\mathcal{F}(a\,\&\,b) &\triangleq \mathcal{F}(a) \vee \mathcal{F}(b)\\
\mathcal{T}(a\,\&\&\,b) &\triangleq \mathcal{T}(a) \wedge \mathcal{T}(b)\\
\mathcal{F}(a\,\&\&\,b) &\triangleq \mathcal{F}(a) \vee \mathcal{F}(b)\\
\mathcal{T}(a\mid b) &\triangleq \mathcal{T}(a) \vee \mathcal{T}(b)\\
\mathcal{F}(a\mid b) &\triangleq \mathcal{F}(a) \wedge \mathcal{F}(b)\\
\mathcal{T}(a\mid\mid b) &\triangleq \mathcal{T}(a) \vee \mathcal{T}(b)\\
\mathcal{F}(a\mid\mid b) &\triangleq \mathcal{F}(a) \wedge \mathcal{F}(b)\\
\mathcal{T}(a\,\texttt{==>}\,b) &\triangleq \mathcal{F}(a) \vee \mathcal{T}(b)\\
\mathcal{F}(a\,\texttt{==>}\,b) &\triangleq \mathcal{T}(a) \wedge \mathcal{F}(b)\\
\mathcal{T}(a\,\texttt{<==}\,b) &\triangleq \mathcal{F}(b) \vee \mathcal{T}(a)\\
\mathcal{F}(a\,\texttt{<==}\,b) &\triangleq \mathcal{T}(b) \wedge \mathcal{F}(a)\\
\mathcal{T}(a\,\texttt{<==>}\,b) &\triangleq (\mathcal{T}(a) \wedge \mathcal{T}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{F}(b))\\
\mathcal{F}(a\,\texttt{<==>}\,b) &\triangleq (\mathcal{T}(a) \wedge \mathcal{F}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{T}(b))\\
\mathcal{T}(a\,\texttt{<=!=>}\,b) &\triangleq (\mathcal{T}(a) \wedge \mathcal{F}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{T}(b))\\
\mathcal{F}(a\,\texttt{<=!=>}\,b) &\triangleq (\mathcal{T}(a) \wedge \mathcal{T}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{F}(b))\\
\mathcal{T}(a\,\hat{}\,b) &\triangleq (\mathcal{T}(a) \wedge \mathcal{F}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{T}(b))\\
\mathcal{F}(a\,\hat{}\,b) &\triangleq (\mathcal{T}(a) \wedge \mathcal{T}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{F}(b))\\
wd_{\mathcal{Y}}(a\,?\,d : e) &\triangleq (\mathcal{T}(a) \wedge wd_{\mathcal{Y}}(d)) \vee (\mathcal{F}(a) \wedge wd_{\mathcal{Y}}(e))\\
&\quad \vee (wd_{\mathcal{Y}}(d) \wedge wd_{\mathcal{Y}}(e) \wedge (d = e))\\
\mathcal{T}(a\,?\,b : c) &\triangleq (\mathcal{T}(a) \wedge \mathcal{T}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{T}(c))\\
&\quad \vee (\mathcal{T}(b) \wedge \mathcal{T}(c))\\
\mathcal{F}(a\,?\,b : c) &\triangleq (\mathcal{T}(a) \wedge \mathcal{F}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{F}(c))\\
&\quad \vee (\mathcal{F}(b) \wedge \mathcal{F}(c))
\end{aligned}
$$

In the following, let furthermore T be a type, i an identifier, which does not appear freely in c and e, i' a fresh identifier, in which i does not appear freely, and a' the counterpart

to a, wherein i is substituted to i'.

$$
\begin{aligned}
wd_{\mathcal{Y}}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(d)\,\texttt{\textbackslash else}(e)) \triangleq\ & \big(\exists\, T\ i\colon \mathcal{T}(a) \wedge wd_{\mathcal{Y}}(d) \\
& \qquad \wedge \forall\, T\ i'\colon ((i' \prec i) \rightarrow \mathcal{F}(a'))\big) \\
& \vee \big(\forall\, T\ i\colon (\mathcal{F}(a) \wedge wd_{\mathcal{Y}}(e))\big) \\
& \vee \big(\forall\, T\ i\colon (wd_{\mathcal{Y}}(d) \wedge wd_{\mathcal{Y}}(e) \wedge (d = e))\big) \\
\mathcal{T}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(b)\,\texttt{\textbackslash else}(c)) \triangleq\ & \big(\exists\, T\ i\colon \mathcal{T}(a) \wedge \mathcal{T}(b) \\
& \qquad \wedge \forall\, T\ i'\colon ((i' \prec i) \rightarrow \mathcal{F}(a'))\big) \\
& \vee (\forall\, T\ i\colon (\mathcal{F}(a) \wedge \mathcal{T}(c))) \\
& \vee (\forall\, T\ i\colon (\mathcal{T}(b) \wedge \mathcal{T}(c))) \\
\mathcal{F}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(b)\,\texttt{\textbackslash else}(c)) \triangleq\ & \big(\exists\, T\ i\colon \mathcal{T}(a) \wedge \mathcal{F}(b) \\
& \qquad \wedge \forall\, T\ i'\colon ((i' \prec i) \rightarrow \mathcal{F}(a'))\big) \\
& \vee (\forall\, T\ i\colon (\mathcal{F}(a) \wedge \mathcal{F}(c))) \\
& \vee (\forall\, T\ i\colon (\mathcal{F}(b) \wedge \mathcal{F}(c))) \\
\mathcal{T}(\texttt{\textbackslash forall}\ T\ i\texttt{;}\ a\texttt{;}\ b) \triangleq\ & \forall\, T\ i\colon \mathcal{T}(a \texttt{==>} b) \\
\mathcal{F}(\texttt{\textbackslash forall}\ T\ i\texttt{;}\ a\texttt{;}\ b) \triangleq\ & \exists\, T\ i\colon \mathcal{F}(a \texttt{==>} b) \\
\mathcal{T}(\texttt{\textbackslash exists}\ T\ i\texttt{;}\ a\texttt{;}\ b) \triangleq\ & \exists\, T\ i\colon \mathcal{T}(a \texttt{\&\&} b) \\
\mathcal{F}(\texttt{\textbackslash exists}\ T\ i\texttt{;}\ a\texttt{;}\ b) \triangleq\ & \forall\, T\ i\colon \mathcal{F}(a \texttt{\&\&} b)
\end{aligned}
$$

Proofs In addition to the original definition, we defined rules for implications, the ternary and the `\ifEx` operator. Whereas implications can be easily deduced by simply taking the equivalent of "$\neg a \vee b$", *well-definedness* of the ternary and the `\ifEx` operator may not be as obvious. Hence, we give proofs to show the deduction of the rules for the $\mathcal{T}$- and the $\mathcal{F}$-operator (in order to supplement the rules in $\mathcal{Y}$-operator semantics) for the ternary and `\ifEx` operator from the *well-definedness* rules for the $\mathcal{D}$-operator.
In the following, for a, b and c expressions of type boolean, we start with the definitions for the $\mathcal{D}$-operator (equivalent semantics as for the $\mathcal{Y}$-operator):

$$
\begin{aligned}
\mathcal{WD}_{\mathcal{Y}}(a\,\texttt{?}\,b : c) &\Leftrightarrow \mathcal{WD}_{\mathcal{D}}(a\,\texttt{?}\,b : c) \\
&\Leftrightarrow (\mathcal{WD}_{\mathcal{D}}(a) \wedge a \wedge \mathcal{WD}_{\mathcal{D}}(b)) \\
&\quad \vee (\mathcal{WD}_{\mathcal{D}}(a) \wedge \neg a \wedge \mathcal{WD}_{\mathcal{D}}(c)) \\
&\quad \vee (\mathcal{WD}_{\mathcal{D}}(b) \wedge \mathcal{WD}_{\mathcal{D}}(c) \wedge (b \leftrightarrow c))
\end{aligned}
\tag{3.1}
$$

$$
\begin{aligned}
\mathcal{WD}_{\mathcal{Y}}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(b)\,\texttt{\textbackslash else}(c)) &\Leftrightarrow \mathcal{WD}_{\mathcal{D}}(\texttt{\textbackslash ifEx}\ T\ i\texttt{;}(a)\,\texttt{\textbackslash then}(b)\,\texttt{\textbackslash else}(c)) \\
&\Leftrightarrow (\exists\, T\ i\colon \mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(b) \wedge a \\
&\qquad \wedge \forall\, T\ i'\colon ((i' \prec i) \rightarrow (\mathcal{WD}_{\mathcal{D}}(a') \wedge \neg a))) \\
&\quad \vee (\forall\, T\ i\colon (\mathcal{WD}_{\mathcal{D}}(a) \wedge \mathcal{WD}_{\mathcal{D}}(c) \wedge \neg a)) \\
&\quad \vee (\forall\, T\ i\colon (\mathcal{WD}_{\mathcal{D}}(b) \wedge \mathcal{WD}_{\mathcal{D}}(c) \wedge (b \leftrightarrow c)))
\end{aligned}
\tag{3.2}
$$

We will also use the following rules from above:

$$\mathcal{T}(a) \Leftrightarrow \mathcal{WD}_{\mathcal{Y}}(a) \wedge a \tag{3.3}$$
$$\mathcal{F}(a) \Leftrightarrow \mathcal{WD}_{\mathcal{Y}}(a) \wedge \neg a \tag{3.4}$$

As well as the definition for `\ifEx` as outlined in the preamble of section 3.1.9:

$$\begin{aligned}\texttt{\ifEx}\ T\ i\texttt{;}(a)\,\texttt{\then}(b)\,\texttt{\else}(c) \Leftrightarrow\ & (\exists\ T\ i\colon a)\\ & \texttt{?}\ (\forall\ T\ i\colon (a \wedge (\forall\ T\ i'\colon a' \rightarrow (i \preceq i'))) \rightarrow b)\\ & \texttt{:}\ c\end{aligned} \tag{3.5}$$

Proof Hence, we can prove "$\mathcal{T}(a\,\texttt{?}\,b : c) \Leftrightarrow (\mathcal{T}(a) \wedge \mathcal{T}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{T}(c)) \vee (\mathcal{T}(b) \wedge \mathcal{T}(c))$".

$$\begin{aligned}
\mathcal{T}(a\,\texttt{?}\,b : c) &\Leftrightarrow \mathcal{WD}_{\mathcal{Y}}(a\,\texttt{?}\,b : c) \wedge (a\,\texttt{?}\,b : c) && \textit{$\mathcal{T}$-operator rule (3.3).}\\
&\Leftrightarrow ((\mathcal{WD}_{\mathcal{Y}}(a) \wedge a \wedge \mathcal{WD}_{\mathcal{Y}}(b))\\
&\quad \vee (\mathcal{WD}_{\mathcal{Y}}(a) \wedge \neg a \wedge \mathcal{WD}_{\mathcal{Y}}(c)) && \textit{$\mathcal{D}$-operator rule (3.1).}\\
&\quad \vee (\mathcal{WD}_{\mathcal{Y}}(b) \wedge \mathcal{WD}_{\mathcal{Y}}(c) \wedge (b \leftrightarrow c)))\\
&\quad \wedge (a\,\texttt{?}\,b : c)\\
&\Leftrightarrow (\mathcal{WD}_{\mathcal{Y}}(a) \wedge a \wedge \mathcal{WD}_{\mathcal{Y}}(b) \wedge (a\,\texttt{?}\,b : c)) && \textit{Expand formula.}\\
&\quad \vee (\mathcal{WD}_{\mathcal{Y}}(a) \wedge \neg a \wedge \mathcal{WD}_{\mathcal{Y}}(c) \wedge (a\,\texttt{?}\,b : c))\\
&\quad \vee (\mathcal{WD}_{\mathcal{Y}}(b) \wedge \mathcal{WD}_{\mathcal{Y}}(c) \wedge (b \leftrightarrow c) \wedge (a\,\texttt{?}\,b : c))\\
&\Leftrightarrow (\mathcal{WD}_{\mathcal{Y}}(a) \wedge a \wedge \mathcal{WD}_{\mathcal{Y}}(b) \wedge b) && \textit{Evaluate expressions}\\
&\quad \vee (\mathcal{WD}_{\mathcal{Y}}(a) \wedge \neg a \wedge \mathcal{WD}_{\mathcal{Y}}(c) \wedge c) && \textit{with tern. operator, e.g.}\\
&\quad \vee (\mathcal{WD}_{\mathcal{Y}}(b) \wedge b \wedge \mathcal{WD}_{\mathcal{Y}}(c) \wedge c) && \text{"}(a \wedge (a\,\texttt{?}\,b : c)) \Leftrightarrow b\text{"}.\\
&\Leftrightarrow (\mathcal{T}(a) \wedge \mathcal{T}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{T}(c)) \vee (\mathcal{T}(b) \wedge \mathcal{T}(c)) && \textit{Rules (3.3) and (3.4).}\ \blacksquare
\end{aligned}$$

Analogously, we deduce "$\mathcal{F}(a\,\texttt{?}\,b : c) \Leftrightarrow (\mathcal{T}(a) \wedge \mathcal{F}(b)) \vee (\mathcal{F}(a) \wedge \mathcal{F}(c)) \vee (\mathcal{F}(b) \wedge \mathcal{F}(c))$".

PROOF We now prove the following equivalence: $\mathcal{T}(\texttt{\ifEx}\ T\ i;(a)\,\texttt{\then}(b)\,\texttt{\else}(c))$

$$\begin{aligned}&\Leftrightarrow\Big(\exists\ T\ i\colon \mathcal{T}(a)\wedge\mathcal{T}(b)\wedge\forall\ T\ i'\colon((i'\prec i)\rightarrow\mathcal{F}(a'))\Big)\\&\quad\vee\Big(\forall\ T\ i\colon(\mathcal{F}(a)\wedge\mathcal{T}(c))\Big)\vee\Big(\forall\ T\ i\colon(\mathcal{T}(b)\wedge\mathcal{T}(c))\Big)\end{aligned}$$

$\mathcal{T}(\texttt{\ifEx}\ T\ i;(a)\,\texttt{\then}(b)\,\texttt{\else}(c))$	
$\Leftrightarrow \mathcal{WD}_{\mathcal{Y}}(\texttt{\ifEx}\ T\ i;(a)\,\texttt{\then}(b)\,\texttt{\else}(c))$	$\mathcal{T}$*-op. rule* (3.3).
$\quad\wedge(\texttt{\ifEx}\ T\ i;(a)\,\texttt{\then}(b)\,\texttt{\else}(c))$	
$\Leftrightarrow\Big(\Big(\exists\ T\ i\colon \mathcal{WD}_{\mathcal{D}}(a)\wedge\mathcal{WD}_{\mathcal{D}}(b)\wedge a$	$\mathcal{D}$*-op. rule* (3.2)
$\qquad\wedge\forall\ T\ i'\colon((i'\prec i)\rightarrow(\mathcal{WD}_{\mathcal{D}}(a')\wedge\neg a'))\Big)$	*and def. for*
$\quad\vee\Big(\forall\ T\ i\colon(\mathcal{WD}_{\mathcal{D}}(a)\wedge\mathcal{WD}_{\mathcal{D}}(c)\wedge\neg a)\Big)$	`\ifEx` *op.* (3.5).
$\quad\vee\Big(\forall\ T\ i\colon(\mathcal{WD}_{\mathcal{D}}(b)\wedge\mathcal{WD}_{\mathcal{D}}(c)\wedge(b\leftrightarrow c))\Big)\Big)$	
$\quad\wedge\Big((\exists\ T\ i\colon a)\,?\,(\forall\ T\ i\colon(a\wedge(\forall\ T\ i'\colon a'\rightarrow(i\preceq i')))\rightarrow b):c\Big)$	
$\Leftrightarrow\Big(\exists\ T\ i\colon \mathcal{WD}_{\mathcal{D}}(a)\wedge\mathcal{WD}_{\mathcal{D}}(b)\wedge a$	*Expand formula.*
$\qquad\wedge\forall\ T\ i'\colon((i'\prec i)\rightarrow(\mathcal{WD}_{D}(a')\wedge\neg a'))$	
$\qquad\wedge\Big((\exists\ T\ i\colon a)\,?\,(\forall\ T\ i\colon(a\wedge(\forall\ T\ i'\colon a'\rightarrow(i\preceq i')))\rightarrow b):c\Big)\Big)$	
$\quad\vee\Big(\forall\ T\ i\colon(\mathcal{WD}_{\mathcal{D}}(a)\wedge\mathcal{WD}_{\mathcal{D}}(c)\wedge\neg a)$	
$\qquad\wedge\Big((\exists\ T\ i\colon a)\,?\,(\forall\ T\ i\colon(a\wedge(\forall\ T\ i'\colon a'\rightarrow(i\preceq i')))\rightarrow b):c\Big)\Big)$	
$\quad\vee\Big(\forall\ T\ i\colon(\mathcal{WD}_{\mathcal{D}}(b)\wedge\mathcal{WD}_{\mathcal{D}}(c)\wedge(b\leftrightarrow c))$	
$\qquad\wedge\Big((\exists\ T\ i\colon a)\,?\,(\forall\ T\ i\colon(a\wedge(\forall\ T\ i'\colon a'\rightarrow(i\preceq i')))\rightarrow b):c\Big)\Big)$	
$\Leftrightarrow\Big(\exists\ T\ i\colon \mathcal{WD}_{\mathcal{D}}(a)\wedge a\wedge\mathcal{WD}_{\mathcal{D}}(b)\wedge b$	*Evaluation of*
$\quad\wedge\forall\ T\ i'\colon((i'\prec i)\rightarrow(\mathcal{WD}_{\mathcal{D}}(a')\wedge\neg a'))\Big)$	*ternary op.*
$\quad\vee\Big(\forall\ T\ i\colon(\mathcal{WD}_{\mathcal{D}}(a)\wedge\neg a\wedge\mathcal{WD}_{\mathcal{D}}(c)\wedge c)\Big)$	
$\quad\vee\Big(\forall\ T\ i\colon(\mathcal{WD}_{\mathcal{D}}(b)\wedge b\wedge\mathcal{WD}_{\mathcal{D}}(c)\wedge c\wedge(b\leftrightarrow c))\Big)$	
$\Leftrightarrow\Big(\exists\ T\ i\colon \mathcal{T}(a)\wedge\mathcal{T}(b)\wedge\forall\ T\ i'\colon((i'\prec i)\rightarrow\mathcal{F}(a'))\Big)$	*Rules* (3.3)
$\quad\vee\Big(\forall\ T\ i\colon(\mathcal{F}(a)\wedge\mathcal{T}(c))\Big)\vee\Big(\forall\ T\ i\colon(\mathcal{T}(b)\wedge\mathcal{T}(c))\Big)$	*and* (3.4). ∎

Analogously, we deduce the according $\mathcal{F}$-operator rule for the `\ifEx` operator.

3.1.10. The `\old` Expression

This expression is a means to make statements in the postcondition, which refer to the pre-state of certain expressions. Let for this matter in the following e be an expression, h the current heap and h_{pre} the heap of the pre-state. In KeY, the `\old`-expression is dealt with by evaluating the argument in the heap of the pre-state, i.e. $\texttt{\old}(h[e]) = h_{pre}[e]$. Thus, it suffices to update the heap of the pre-state to the current heap, as a parallel *update* $\{h_{pre} \coloneqq h\}$ to the *anonymisation update* (section 2.7.1) for the whole postcondition.

As the *anonymisation update* only updates expressions in heap h, it does not apply to expressions in heap h_{pre} and hence in each case only the appropriate *update* gets applied. This leads us to the following definition:

$$wd(\texttt{\textbackslash old}(h[e])) \triangleq wd(h_{pre}[e])^1$$

3.1.11. Invariant References

In JML*, non-static invariant references are denoted as $o.\texttt{\textbackslash inv}$ for some object o, whereas in original JML the according expression is `\invariant_for(`o`)`.
They are essentially syntactic sugar defining the *represents*-clause of the model field `\inv` (the actual invariant). In proofs, they are used as any other *represents*-clause. However, there are two different versions of them, namely a static and a non-static one. As such, their *well-definedness* rules basically work as they do for any other reference expression.

3.1.11.1. Non-Static

Let for the following definition a be an expression of non-static reference type and h a heap.

$$wd(h[a.\texttt{\textbackslash inv}]) \triangleq wd(a) \land wd(h) \land (a \neq \texttt{null}) \land h[a.created] \land wellFormed(h)$$

3.1.11.2. Static

Let furthermore below h be a heap and C a static class.

$$wd(h[C.\texttt{\textbackslash inv}]) \triangleq wd(h) \land wellFormed(h)$$

3.1.12. Pure Method Invocations

As we do not desugar method contracts combined by `also` into one method contract, one invocation might refer to several method contracts. For this matter, we understand $\textbf{\textit{requires}}_m$ to denote the disjunction of all the preconditions – of contracts not allowing m to cast an exception – specified for a method m. If there are only contracts allowing the cast of exceptions for a method m, we take the disjunction to be equal to *false* (i.e. an invocation of m is hence not *well-defined*).

[1] The *update* $\{h_{pre} := h\}$ is applied to the whole postcondition and thus also applies to all expressions using the `\old` expression (instead of the *anonymisation update*).

3.1.12.1. Non-Static

Let for the following definition a be an expression of reference type, h a heap, $e_1, \ldots, e_n$ the parameters and m the according method identifier of a pure (non-void) non-static method.

$$\begin{aligned} wd(h[a.m(e_1, \ldots, e_n)]) \triangleq\ & wd(a) \wedge wd(h) \wedge (a \neq \texttt{null}) \wedge h[a.created] \\ & \wedge\ wellFormed(h) \wedge h[a.\,\texttt{\textbackslash inv}] \\ & \wedge\ \forall i \in [1, n]\colon wd(e_i) \wedge \textbf{\textit{requires}}_m \end{aligned}$$

3.1.12.2. Static

Let furthermore below h be a heap, $e_1, \ldots, e_n$ the parameters and m the according method identifier of a pure (non-void and non-constructor) static method of class C.

$$\begin{aligned} wd(h[C.m(e_1, \ldots, e_n)]) \triangleq\ & wd(h) \wedge wellFormed(h) \wedge h[C.\,\texttt{\textbackslash inv}] \\ & \wedge\ \forall i \in [1, n]\colon wd(e_i) \wedge \textbf{\textit{requires}}_m \end{aligned}$$

3.1.12.3. Constructor

Let moreover below h be a heap, $e_1, \ldots, e_n$ the parameters and m the according method identifier of a pure constructor of type T.

$$wd(h[\texttt{new}\ T_m(e_1, \ldots, e_n)]) \triangleq wd(h) \wedge wellFormed(h) \wedge \forall i \in [1, n]\colon wd(e_i) \wedge \textbf{\textit{requires}}_m$$

3.1.13. Array Creations

The specification of array creations as defined in JML is currently not supported by KeY. However, an equivalent specification can be done using the sequence data type in JML*.

3.1.13.1. Initialiser

Let for the following definition $e_0, \ldots, e_{n-1}$ be expressions of comparable types.

$$wd(\{e_0, \ldots, e_{n-1}\}) \triangleq \forall i \in [0, n-1]\colon wd(e_i)$$

3.1.13.2. Declaration

Let furthermore below T be a type, $n_1, \ldots, n_k$ expressions of type int, $\underline{init}$ an array initialiser of type $T[]^k$ (result type $T[] \ldots []$ for a k-dimensional array).

$$wd(\texttt{new}\ T[n_1][n_2] \ldots [n_k]\ \underline{init}) \triangleq wd(\underline{init}) \wedge \forall i \in [1, k]\colon (wd(n_i) \wedge n_i \geq 0)$$

3.2. JML Specifications

JML specifications are used to annotate syntactical entities in a Java program and describe their behaviour. These entities can be either a whole class, a method or a model field. Whereas a class or a method[1] also matters in the execution of the program, model fields are specification-only. We give general (desugared) blueprints and definitions for *well-definedness* rules for class invariants, model fields and method contracts below.

3.2.1. Class Invariants

```
/*@ V k invariant invariant_C
  @  accessible \inv:  accessible_C
  @  \measured_by measured-by_C;
  @*/
```

Figure 3.1.: Class Invariant Specification for a Class C

Let for this definition C be a class identifier, V a visibility, k one of the keywords `static` or `instance` and $\mathit{invariant}_C$ the (with respect to some linear type hierarchy) constructed short-circuit conjunction of all *invariants* defined in super-types (including interfaces) of C (including C itself) and inherited according to Java rules. This basically means that when taking an invariant, we also must consider the inherited invariants, which – for the matter of *well-definedness* – guard invariants in subclasses.
As in the context of *well-definedness* the following clauses are all treated in a similar manner, we denote the set $U_C := \{\mathit{accessible}_C, \mathit{measured\_by}_C\}$ for the following definition.

$$wd(C) \triangleq \mathcal{WD}(\mathit{invariant}_C) \wedge (\mathit{invariant}_C \rightarrow \bigwedge_{u \in U_C} wd(u))$$

3.2.2. Model Fields

```
/*@ model T m
  @  requires requires_m;
  @  measured_by measured-by_m;
  @  accessible accessible_m;
  @  represents represents_m;
  @*/
```

Figure 3.2.: Model Field Specification for a Model Field m

Let for the following definition m be the model field identifier for a model field of type T.

[1] Methods can also be specification-only, being specified by the keyword `model_behaviour` (not standard JML), but will not be considered in this work as their semantics is still being discussed.

As in the context of *well-definedness* the following clauses are all treated in a similar manner, we denote the set $U_m \coloneqq \{\texttt{accessible}_m, \texttt{represents}_m, \texttt{measured-by}_m\}$ for the rule below.

$$wd(m) \triangleq \mathcal{WD}(\texttt{requires}_m) \wedge (\bigwedge_{u \in U_m} wd(u))$$

3.2.3. Method Contracts

```
/*@ V behaviour
  @  requires requires_m;
  @  measured_by measured-by_m;
  @  diverges diverges_m;
  @  accessible accessible_m;
  @  assignable assignable_m;
  @  ensures ensures_m;
  @  signals signals_m;
  @*/
```

Figure 3.3.: Method Contract Specification for a Method m

Let for the following definition m be the method identifier for a method being specified in class C and let $\{anon^{\texttt{assignable}_m}\}$ be the update anonymising the heap in all locations specified in the *assignable*-clause $\texttt{assignable}_m$.
Let furthermore $\texttt{invariant}_C$ be the (with respect to some linear type hierarchy) constructed short-circuit conjunction of all *invariants* defined in super-types (including interfaces) of C (including C itself) and inherited according to Java rules (as in section 3.2.1) and $\texttt{initially}_C$ be constructed in a similar fashion using the *initially*-clauses instead of *invariants*.
As in the context of *well-definedness* the following clauses are all treated in a similar manner, we denote the set $U_m \coloneqq \{\texttt{accessible}_m, \texttt{assignable}_m, \texttt{diverges}_m, \texttt{measured-by}_m\}$ for the rule below.

$$\begin{aligned} wd(m) \triangleq\ & \mathcal{WD}(\texttt{requires}_m) \\ & \wedge (\texttt{requires}_m \\ & \quad \rightarrow (\bigwedge_{u \in U_m} wd(u) \\ & \qquad \wedge \{anon^{\texttt{assignable}_m}\} \\ & \qquad\quad (\mathcal{WD}(\texttt{initially}_C{}^{1} \,\&\&\, \texttt{ensures}_m) \\ & \qquad\quad \wedge \mathcal{WD}(\texttt{initially}_C{}^{1} \,\&\&\, \texttt{signals}_m)))) \end{aligned}$$

[1] Only for (non-helper) constructor methods.

3.3. JML Statements

In the following, JML statements will be discussed. They are specification elements which can appear directly in the Java code. We give general (desugared) blueprints, reminders of the verification rules and definitions for *well-definedness* rules for loop statements, which annotate loops, and block contracts, which annotate blocks. Whereas loop statements are widely used in formal verification, blocks are a rather new feature implemented in KeY since 2012 [Wac12].
A block is a sequence of no, one or multiple Java instructions between curly braces. It can appear anywhere, where an instruction is legal. Block contracts are similar to method contracts, but supplementary permit to specify further types of abrupt termination via *breaks*-, *continues*- and *returns*-clauses.
Since these statements can be seen as a drawback to our black-box assumption (section 2.6), they are dealt with "on the fly", i.e. when appearing in the code. In fact, we implemented their *Well-Definedness Checks* as part of the ordinary verification rules for loop statements and block contracts, because these statements are always *context*-dependent, meaning they are written in a specific state of the method in which they are embedded. Thus, we define their pre-state relative to their method's pre-state by applying a *context-update* $\{u\}$ on it.

3.3.1. Loop Statements

```
/*@ loop_invariant loop-invariant_l;
  @  assignable assignable_l;
  @  decreases decreases_l;
  @*/
```

Figure 3.4.: Loop Statement Specification for a Loop l

The *Well-Definedness Check* of loop statements is embedded in the general rule for loop invariants as a new fourth branch as shown in definition 3, where l identifies the loop and $\{u\}$ denotes a context-update embodying all changes between the pre-state of the method and entering the loop. We hence define the new rule as follows.

Definition 3 (Loop Invariant Rule with *Well-Definedness* Branch)

$$\frac{\begin{array}{c}\text{Initially Valid}\\ \text{Body Preserves Invariant}\\ \text{Use Case}\\ \Gamma \Rightarrow \{u\}wd(l), \Delta \ \textit{(}\text{Well-Definedness}\textit{)}\end{array}}{\Gamma \Rightarrow \{u\}[\pi\ \texttt{while}_l(g)p\ ;\ \omega]\varphi, \Delta}$$

Let $\{anon^{\texttt{assignable}_l}\}$ be the update anonymising the heap in all locations specified in the *assignable*-clause $\texttt{assignable}_l$ for the definition below.

$$
\begin{aligned}
wd(l) \triangleq\ & wd(\texttt{loop-invariant}_l) \\
& \wedge (\texttt{loop-invariant}_l \\
& \quad \rightarrow (wd(\texttt{assignable}_l) \wedge wd_s(\texttt{decreases}_l) \\
& \quad\quad \wedge \{anon^{\texttt{assignable}_l}\}(wd(\texttt{loop-invariant}_l) \wedge wd(\texttt{decreases}_l))))
\end{aligned}
$$

3.3.2. Block Contracts

```
/*@  requires requires_b;
  @  diverges diverges_b;
  @  assignable assignable_b;
  @  ensures ensures_b;
  @  signals signals_b;
  @  breaks breaks_b;
  @  continues continues_b;
  @  returns returns_b;
  @*/
```

Figure 3.5.: Block Contract Specification for a Block b [Wac12]

The *Well-Definedness Check* of block contracts is embedded in the general rule for block contracts as a new fourth branch as shown in definition 4, where b identifies the block and $\{u\}$ denotes a context-update embodying all changes between the pre-state of the method and entering the block. We hence define the new rule as follows.

Definition 4 (Block Contract Rule with *Well-Definedness* Branch)

$$
\frac{\begin{array}{c}\text{Validity} \\ \text{Precondition} \\ \text{Usage} \\ \Gamma \Rightarrow \{u\} wd(b), \Delta \ \textit{(}\text{Well-Definedness}\textit{)}\end{array}}{\Gamma \Rightarrow \{u\}[\pi\{\texttt{block}_b\}\ ;\ \omega]\varphi, \Delta}
$$

Let $\{anon^{\texttt{assignable}_b}\}$ be the update anonymising the heap in all locations specified in the *assignable*-clause $\texttt{assignable}_b$. For block contracts, we denote the sets $U_b := \{\texttt{assignable}_b, \texttt{diverges}_b\}$ and $POST_b := \{\texttt{ensures}_b, \texttt{signals}_b, \texttt{breaks}_b, \texttt{continues}_b, \texttt{returns}_b\}$ to define the following.

$$
\begin{aligned}
wd(b) \triangleq\ & wd(\texttt{requires}_b) \\
& \wedge (\texttt{requires}_b \rightarrow (\bigwedge_{u \in U_b} wd(u) \wedge \{anon^{\texttt{assignable}_b}\}(\bigwedge_{p \in POST_b} wd(p))))
\end{aligned}
$$

4. Integration in the KeY System

An essential part of this work is the implementation of *Well-Definedness Checks* in the KeY System. All the definitions and rules stated in this work are hereby considered to result in fully implemented *Well-Definedness Checks* for the major part of the JML/JML* specification elements supported by KeY. The implementation is taclet[1]-driven, which results in a flexible, adaptable and extensible mechanism to cope with KeY's vast functionality and its fast and active development.

4.1. Design Decisions

As KeY's major focus is the verification of JML*-annotated Java programs, we needed to make some adaptations and adjustments to integrate fully functional *Well-Definedness Checks*. The main reasons are certain syntactical characteristics. Firstly, expression order does not matter in the ordinary proof process, but can be crucial for the result of a *Well-Definedness Check* (only in $\mathcal{L}$-operator semantics, section 2.5.2). Secondly, a different kind of operation, a *term transformer* as stated in section 2.5, is needed as the evaluation done by $\mathcal{WD}$ and *wd* highly differs from the usual evaluation rules in KeY. For example, "$a \vee \neg a$" does not necessarily have the same result as "*true*" in a *Well-Definedness Check*.

4.1.1. Short-Circuit Operators

As verification in KeY operates in predicate logic, there is no operator for short-circuit evaluation (e.g. `&&` as opposed to `&`) and thus no implementation of it. However, the semantics of the $\mathcal{L}$-operator necessitates operators which distinguish the order of its arguments. Subsequently, an implementation of short-circuit evaluation becomes necessary. Up to now, these do only matter for *Well-Definedness Checks* and are irrelevant in the ordinary verification process. As we do not want to irritate KeY users by presenting them new operators with no apparent difference to their non-short-circuit versions, we directly label the according terms built with short-circuit operators (e.g. "`a && b`") with a term label[2] `«SC»` for short-circuit operators. In the KeY System, these labels can be automatically hidden or made visible by the choice of the user. However, they can be

[1] Taclets are lightweight entities to logically and technically define calculus rules [Gie04].

[2] Term labels are a quite recent and purely syntactical feature in KeY, being a means to enhance any term with supplementary information. Apart from *Well-Definedness Checks*, they are mainly used to guide verification strategies and enhance performance.

used for matching in rules of the sequent calculus. In terms of simplification, a formula "$a \leftarrow b$" thus becomes "$(a \vee \neg b)$«SC»".

4.1.2. Order in Specification Clauses

Now having short-circuit operators, the order of formulas in con- and disjunctions hence becomes important. This might be obvious for the explicit parts of the specification, but certain characteristics of JML also call for implicit specification elements. One characteristic being for example that all elements are non-null by default, including array elements. As this property also must be expressed by formulas, this has to be added to the specification in the process of desugaring. Up to now, formula order was not an issue in KeY and since different characteristics of the specified target necessitate different formulas to be added, a consistent ordering from the beginning might perturb the rest of KeY. Again, we use the means of term labels, which can be searched for fast and easily, for ordering the specification clauses prior to checking the specification's *well-definedness*. The label «impl» thus denotes implicit specification elements.

4.1.3. Term Transformers

Our notion of *well-definedness* also creates the need for new operators, which leave the expression uninterpreted, as in the rest of the KeY System the semantics differs (section 2.5). This cannot be considered a usual function and thus needs some adjustments to fit in the KeY System. To cope with this deficiency, we introduced a new kind of operation named *term transformer* in KeY, which suffices our needs. The concept of this operator is based on Dijkstra's *predicate transformer semantics* [Dij75]. A more conceptual description of *term transformers* is given in section 2.5.

4.2. Rule Implementation

As the majority of rules in the KeY System is implemented as so-called *taclets* [Rüm07], we implemented the *well-definedness* calculus likewise. This is except for "complex" JML specifications (at present class invariants, model fields, method contracts, loop statements and block contracts) as these form the embedding proof obligations for our *Well-Definedness Checks*. The majority of these taclets can be implemented in a target- and proof-independent way, leaving us only with the rules for invariant references and pure method invocations to be generated based on the annotated java code. Regarding the taclets for method invocations, this is based on the need of the specific preconditions, whereas for the taclets for invariant references, this is actually independent of any given annotated java code, but as the type `java.lang.Object` is not available before loading the KeY System, this taclet also has to be generated upon initialisation.
For the taclet mechanism in KeY being already quite powerful, only four *variable conditions* [Rüm07] needed to be implemented. Firstly, this is an "atomic condition" to identify expressions with no further sub-arguments, as these are always *well-defined*.

Secondly, this is a "static field condition" to check if a field is static or an instance, as only for non-static fields a real receiver object exists and can be checked for equality to `null`. Thirdly, this is a "sub-formula condition" checking whether a formula has any sub-arguments which are formulas (as opposed to being a term). This condition is important for the implementation of the $\mathcal{Y}$-operator, having different transformer predicates, which needs a condition to descend into $\mathcal{T}$ and $\mathcal{F}$ or not, as the majority of the *well-definedness* rules, which take a term as argument, stays the same and hence does not have to be defined again for the $\mathcal{Y}$-operator. Fourthly and finally, this is a "term label condition" to check whether a matched expression contains a specific term label or not. For the matter of this implementation, we need it only to distinguish `&&` (`||`) and `&` (`|`) respectively (section 4.1.1).

4.3. Short Examples

In the following, we will perform the implemented *Well-Definedness Check* on two short examples to demonstrate its usage and behaviour in the KeY System. Furthermore, these examples illustrate the differences of *well-definedness* in JML/JML* to *well-definedness* in Java, thereby introducing two new exception types which do only relate to expressions in JML/JML*, but not in Java.

4.3.1. Example for Missing Invariant

```
class A {
        /*@ normal_behaviour
          @ requires
          @           this.\inv;
          @*/
        /*@pure@*/ int id() {
                return 0;
        }

        /*@ ensures
          @          \result.id() == 0;
          @*/
        A m() {
                return new A();
        }
}
```

Listing 4.1: Example Specification One

Let us for example take the specification in listing 4.1 for a class `A` with the two methods `id()` and `m()` for an arbitrary invariant as a simple example for performing a *Well-Definedness Check* for the method `m()`. The method `id()` is specified to be pure, terminate normally and not throw any exceptions. As in JML/JML*, declarations are non-null by default, a null-check for the result of `m()` is not necessary, being already implicitly specified. In JML/JML*, only pure methods should be invoked in specifications. This does not matter for *well-definedness*, but invocations of non-pure methods can still lead to unexpected results. But what does matter to *well-definedness* is that invoked methods are specified to terminate normally and not throw any exceptions, as this is one of the main goals of the *Well-Definedness Check*.

```
==>
   WD(    wellFormed(heap)
       && !self = null
       && self.<created> = TRUE
       && A::exactInstance(self) = TRUE
       && self.<inv>)
 & ( (   wellFormed(heap)
      & !self = null
      & self.<created> = TRUE
      & A::exactInstance(self) = TRUE
      & self.<inv>
      & wellFormed(anon_heap))
  -> (  wd(allLocs)
      & wd(allLocs)
      & {heapAtPre:=heap
         || heap:=anon(heap, allLocs, anon_heap)}
           WD(  (result = null
                    || result.<created> = TRUE)
                &&   (exc = null
                    -> (self.<inv>
                      && !result = null
                      && result.id() = 0))
                && (   !exc = null
                       -> ((java.lang.Exception
                                ::instance(exc) = TRUE
                            -> self.<inv>)
                          && java.lang.RuntimeException
                                ::instance(exc) = TRUE)))))
```

Listing 4.2: Example Proof Obligation One

Performing the *Well-Definedness Check* in KeY for the method `m()`, we obtain the proof obligation in listing 4.2 (to facilitate reading it, we substituted the label "`«SC»`" by the

operator "`&&`" or "`||`" accordingly). The update "`{heapAtPre:=heap}`" is for dealing with the JML operator "`\old`" (section 3.1.10) (which plays no role in this example specification though) and the update "`{heap:=anon(heap, allLocs, anon_heap}`" is the *anonymisation update* as discussed in section 2.7.1. The remaining proof obligation is as defined in section 3.2.3, also including implicit assumptions and specifications such as the invariant reference in the pre- and postcondition, the non-null and created statements, the *signals*-clause for run-time exceptions and the assumption of *well-formed* heaps. We will now apply *well-definedness* rules as well as various simplification rules in the KeY System.

```
 exc = null,
 result.<created> = TRUE
     | anon_heap[result.<created>] = TRUE,
 wellFormed(heap),
 wellFormed(anon_heap),
 A::exactInstance(self) = TRUE,
 self.<created> = TRUE,
 anon(heap, allLocs, anon_heap)[self.<inv>],
 self.<inv>
==>
 result = null,
 self = null,
 anon(heap, allLocs, anon_heap)[result.<inv>]
```

Listing 4.3: Open Goal of Example Proof Obligation One

This evaluates in the open goal illustrated in listing 4.3 and thus we investigate for a counterexample, for which the specification of method `m()` is not *well-defined*. To make its specification *well-defined*, we need to prepend "`\result.\inv &&`" to its postcondition. By looking at the open goal from listing 4.3, this can be easily deduced upon spotting "`anon(heap, allLocs, anon_heap)[result.<inv>]`" in the succedent.
The reason is that before invoking a method, its class invariant (in this case the instance invariant for its receiver) has to be valid. Without this being the case, the method contract is not defined and thus cannot be applied. In this work, we call this problem an `UndefinedMethodInvocation`. This is something that cannot happen in usual Java programs as there are no specifications and the Java code is taken instead.

4.3.2. Example for Uncreated Object

```
class B {
        /*@ normal_behaviour
          @ ensures (\forall Object o; o != null;
          @                  \old(f(o)) == f(o));
          @*/
        /*@pure@*/ void n() {
            Object o1 = new Object();
            Object o2 = new Object();
        }

        /*@ normal_behaviour
          @ ensures true;*/
        /*@pure@*/ int f(Object o) {
            if (o == null) {
                return 0;
            } else {
                return 1;
            }
        }
}
```

Listing 4.4: Example Specification Two

Let us now take the specification in listing 4.4 for a class `B` with the two methods `n()` and `f(Object)` for an arbitrary invariant as another simple example for performing a *Well-Definedness Check*. The two methods are both specified to be pure, terminate normally and not throw any exceptions. In the postcondition for `n()`, a quantification over objects which are not equal to `null` is used as well as the `\old` expression and an invocation of method `f(Object)`, which has a trivial method contract. We will now perform a *Well-Definedness Check* for the method `n()`.

```
==>
   WD(    wellFormed(heap)
       && !self = null
       && self.<created> = TRUE
       && B::exactInstance(self) = TRUE
       && self.<inv>)
 & (     wellFormed(heap)
       && !self = null
       && self.<created> = TRUE
       && B::exactInstance(self) = TRUE
       && self.<inv>
       & wellFormed(anon_heap)
    ->   wd(allLocs)
       & wd(allLocs)
       & {heapAtPre:=heap
          || heap:=anon(heap, allLocs, anon_heap)}
          WD(  exc = null
             & (  self.<inv>
                && \forall java.lang.Object o;
                     (   !o = null
                       & (o = null || o.<created> = TRUE)
                     -> heapAtPre[self.f(o)] = self.f(o)))))
```

Listing 4.5: Example Proof Obligation Two

Performing the *Well-Definedness Check* in KeY for the method `n()`, we obtain the proof obligation in listing 4.5 (to facilitate reading it, we substituted the label "`«SC»`" by the operator "`&&`" or "`||`" accordingly). The update "`{heapAtPre:=heap}`" is for dealing with the JML operator "`\old`" (section 3.1.10) and the update "`{heap:=anon(heap, allLocs, anon_heap}`" is the *anonymisation update* as discussed in section 2.7.1. The remaining proof obligation is as defined in section 3.2.3, also including implicit assumptions and specifications such as the invariant reference in the pre- and postcondition, the non-null and created statements and the assumption of *well-formed* heaps. In this example, the update for dealing with the `\old` expression is actually needed as this expression occurs in `n()`'s postcondition. We will now apply *well-definedness* rules as well as various simplification rules in the KeY System.

```
 anon_heap[o_0.<created>] = TRUE,
 wellFormed(heap),
 wellFormed(anon_heap),
 B::exactInstance(self) = TRUE,
 self.<created> = TRUE,
 self.<inv>,
 anon(heap, allLocs, anon_heap)[self.<inv>]
==>
 self = null,
 o_0.<created> = TRUE,
 o_0 = null
```

Listing 4.6: Open Goal of Example Proof Obligation Two

This evaluates in the open goal illustrated in listing 4.6 and thus we investigate for a counterexample, for which the specification of method `n()` is not *well-defined*. To make its specification *well-defined*, we need to conjunct "`!\fresh(o)`", specifying that `o` was already created in the pre-state, to the range predicate of the universal quantifier in its postcondition. By looking at the open goal from listing 4.6, this can be easily deduced upon spotting "`anon_heap[o_0.<created>] = TRUE`" in the antecedent, but "`o_0.<created> = TRUE`" in the succedent.
The reason is that when making any statement about an object, the element needs to be declared first. This means it either needs to be created or equal to `null`. As in the postcondition, we quantify over a set of objects and also refer to them in the pre-state (by using the `\old` expression) and include all those in a statement as argument for `f(Object)`, this results in an `ObjectNotCreatedException`, as we call it in this work. This is again something that cannot happen in usual Java programs as there, only statements about reachable (i.e. objects which are either equal to `null` or are already created) objects can be made. But due to JML/JML*'s state semantics, this can lead to statements about the *undefined*.

5. Case Study

Within this work, we perform a case study on the JML specification elements for the main part of the KeY examples, which are included in the KeY System. Therein, we checked the *well-definedness* of more than 70 different Java files, many of them containing an invariant, multiple different method contracts, model fields, loop statements and block contracts. Most of them turned out to be *well-defined*. In terms of performing *Well-Definedness Checks* on those specification elements, we will mention some found to be *ill-defined* under the $\mathcal{L}$-operator[1] and discuss their problems. Subsequently, we will conclude this chapter by discussing the performance of the implemented *Well-Definedness Check* for a subset of the checked method specifications.

5.1. Ill-Defined Specification Elements

Based on the types of `Java Unchecked RuntimeException`, we classified the *ill-defined* specification elements to the according exception type. JML* is indeed very similar to Java, but also differs in some points and thus we detected some *definedness* problems for JML* specification elements, which are not covered by the Java exception types. Consequently, we introduced two new exception types to tackle these issues. For the matter of pure method invocations, we introduced a new exception type named `UndefinedMethodInvocation`, meaning some parts of the invoked method's precondition (including the receiver's class invariant) are not guaranteed to hold upon invoking it. Furthermore, we defined a new exception type named `ObjectNotCreatedException`, meaning that a statement is made about an object that is not created yet. We will now discuss the specification elements found to be *ill-defined* under the $\mathcal{L}$-operator.

5.1.1. Index Out Of Bounds Exception

This exception type denotes that some type of index is out-of-bounds. It also includes the inherited exception `ArrayIndexOutOfBoundsException` stating that an array index is out-of-bounds. More specifically, this means that an indexed data structure is accessed at an index position which lies either below the smallest or above the biggest index for which it contains elements. For arrays and many other indexed data structures, the smallest index for which it contains an element is zero. The biggest index can usually be computed by using some kind of `length` function.

[1] This choice was made purely for performance reasons and we will explicitly note, when specification elements are only not *well-defined* in this semantics, but are *well-defined* in the other semantics.

5.1.1.1. Methods in `ReverseArray`

```
public class ReverseArray {
    public int[] a;
    ...
    /*@ public normal_behavior
      @ requires a!=null && a.length>=0;
      @ ensures (\forall int j; j>=0 && j<a.length;
      @             a[j]==\old(a[a.length-(j+1)]));
      @ diverges true;
      @*/
    public void reverse() { ... }

    /*@ public normal_behavior
      @ requires p_a!=null && p_a.length>=0;
      @ ensures (\forall int j; j>=0 && j<\old(p_a.length);
      @            \result[j]==\old(p_a[p_a.length-(j+1)]))
      @      && \result.length == \old(p_a.length);
      @ diverges true;
      @*/
    public int[] reverse2(int[] p_a) { ... }
}
```

Listing 5.1: Specifications for `ReverseArray`

We start with the class `ReverseArray` for our purpose containing the two methods `reverse()` and `reverse2(int[])`, both specified to not throw any exceptions. The objective of these methods is the intuitive one, each reversing an array. For `reverse()`, this is the array `a`, a global variable in `ReverseArray`, and `reverse2(int[])` reverses the array `p_a` given as a parameter.
The specification for `reverse()` in listing 5.1 is not *well-defined* because of its postcondition. Therein, the pre-state of the array `a` is accessed at the position "`a.length` $-(j+1)$", although j is only bound relative to the post-state of `a`. It can very well be greater or equal than `a.length` though, which would result in an undefined access to `a`.
The specification for `reverse2()` in listing 5.1 is not *well-defined* under the $\mathcal{L}$-operator, though under the $\mathcal{D}$-operator it is. The reason is that in the quantified formula the variable j is only bound relative to `p_a.length` and not `\result.length`. For also being *well-defined* under the $\mathcal{L}$-operator, the two parts of the conjunction in the postcondition would need to be interchanged.

5.1.1.2. Method in `BinarySearch`

```
class BinarySearch {
    /*@ public normal_behaviour
      @  requires (\forall int x;
      @   (\forall int y; 0 <= x && x < y && y < a.length;
      @   a[x] <= a[y]));
      @  ensures ((\exists int x; 0 <= x && x < a.length;
      @                 a[x] == v) ?
      @       a[\result] == v : \result == -1);
      @*/
    static /*@pure@*/ int search(int[] a, int v) { ... }
}
```

Listing 5.2: Specification for `BinarySearch.search(int[],int)`

The class `BinarySearch` contains exactly one method. This method is static and specified to not have any side-effects and to not throw any exceptions. Its objective is to take a sorted array `a` of exact numerical values and a single exact numerical value `v`, and to return the index of `v` if it exists in `a` or -1 if it does not.
This specification in listing 5.2 is not *well-defined* because of its postcondition. The problem lies in the expression `a[\result]`, where `\result` is not guarded by any means and thus can easily be less than zero or greater or equal to `a.length`.

5.1.1.3. Method in `LCP`

```
final class LCP {
    /*@ normal_behavior
      @ requires 0 <= x && x < a.length;
      @ requires 0 <= y && y < a.length;
      @ requires x != y;
      @ ensures 0 <= \result;
      @ ensures \result <= a.length-x
      @  && \result <= a.length-y;
      @  && (\forall int i; 0 <= i && i < \result;
      @                     a[x+i] == a[y+i]);
      @ ensures a[x+\result] != a[y+\result]
      @       || \result == a.length-x
      @       || \result == a.length-y;
      @ strictly_pure @*/
    static int lcp(int[] a, int x, int y) { ... }
}
```

Listing 5.3: Specification for `LCP.lcp(int[],int,int)`

The class `LCP` contains one method `lcp(int[],int,int)`. This method is static and specified to be strictly pure, terminate normally and not throw any exceptions. The keyword `strictly_pure` is not standard JML, but part of JML*. It specifies the method to be pure, i.e. not have any side-effects, and as well not to create any new objects on the heap. This means that the heap in the pre-state is exactly the same as the one in the post-state. The method's objective is to take some array `a` of exact numerical values as well as two exact numerical values `x` and `y`, which are not equal and inside the bounds of `a`. Then the method searches in the elements of `a` for the index i up to which the element with the index $x + i$ and the index $y + i$ is equal, i.e. the length of the longest common prefix.
This specification in listing 5.3 is not *well-defined* under the $\mathcal{L}$-operator, though under the $\mathcal{D}$-operator it is. The reason is that in the last part of the postcondition, the array `a` gets accessed in position "$x +$ `\result`" and "$y +$ `\result`". At this moment, the expression `\result` is only bound to be greater or equal to zero and less or equal to "`a.length` $-x$" and "`a.length` $-y$". However, if `\result` was equal to either of those, this would result in `a` being accessed at position `a.length`, an undefined access.

5.1.1.4. Invariant for `SuffixArray`

```
public final class SuffixArray {
    /*@ public invariant
      @     (\forall int i; 0 <= i && i < a.length;
      @          (\exists int j; 0 <= j && j < a.length;
      @                          suffixes[j]==i));
      @  && (\forall int i; 0 <= i && i < a.length;
      @         0 <= suffixes[i] && suffixes[i] < a.length);
      @  && (\forall int i; 0 < i && i < a.length;
      @                       suffixes[i-1] != suffixes[i]);
      @  && (\forall int i; 0 < i && i < a.length;
      @          compare(suffixes[i],suffixes[i-1]) > 0);
      @  && a.length == suffixes.length;
      @*/
    ...
    final /*@ spec_public @*/ int[] a;
    final /*@ spec_public @*/ int[] suffixes;
}
```

Listing 5.4: Specification for `SuffixArray`

The class `SuffixArray` implements a data structure of the two arrays `a` and `suffixes`, both containing exact numerical values. It also offers a method `compare(int, int)` which lexicographically compares two integers with respect to the result of the previously specified function `LCP.lcp(int[], int, int)` for the array `a`. The array `suffixes` is the lexicographically sorted (by using `compare(int, int)`) version of `a`.

The specification in listing 5.4 is not *well-defined* under the $\mathcal{L}$-operator as `suffixes[j]` might throw an `IndexOutOfBoundsException` for `i` could be outside the bounds of `suffixes`. However, this is not the case for the $\mathcal{D}$-operator, since there the assertion "`a.length == suffixes.length`" ensures its *well-definedness*.

5.1.1.5. Method in `Saddleback`

```
class Saddleback {
    /*@ public normal_behaviour
      @ requires (\forall int i; 0<=i && i<array.length;
      @              array[i].length == array[0].length);
      @   && array.length > 0 && array[0].length > 0;
      @   && (\forall int k,i,j; 0 <= k && k < i
      @        && i < array.length && 0<=j
      @        && j < array[0].length;
      @            array[k][j] <= array[i][j]);
      @   && (\forall int k,j,i; 0 <= i && i < array.length
      @        && 0<=k && k<j && j < array[i].length;
      @            array[i][k] <= array[i][j]);
      @ ensures \result == null ==>
      @     (\forall int i; 0<=i && i<array.length;
      @       (\forall int j; 0<=j && j<array[i].length;
      @         array[i][j] != value));
      @ ensures \result != null ==>
      @     \result.length == 2 &&
      @     array[\result[0]][\result[1]] == value;
      @ assignable \nothing;
      @*/
    public /*@nullable*/ int[]
           search(int[][] array, int value) { ... }
}
```

Listing 5.5: Specification for `Saddleback.search(int[][],int)`

The class `Saddleback` contains one method `search(int[][], int)`. It is specified to terminate normally, not throw any exceptions and not have any side-effects (here specified by the keyword `\nothing` in the assignable-clause). This method takes a two-dimensional array of exact numerical values and a single exact numerical value. The array's sub-arrays need to be of equal length bigger than zero, and the values along any row or column are non-decreasing. Then the method returns either the two indices `x` and `y` for which `array[x][y]` is equal to the given value, or `null` if the value does not exist in the array. The specification in listing 5.5 is not *well-defined* as there is nothing known about `\result[0]` and `\result[1]` except that both do exist and hold a value of type `int`. Hence, the expression "`array[\result[0]][\result[1]]`" is highly fragile, potentially throwing an `IndexOutOfBoundsException` or a `NullPointerException`. Firstly,

`\result[0]` might be less than zero or bigger than `array.length`, potentially throwing an `IndexOutOfBoundsException`. Secondly, if there was no exception yet, the result of `array[\result[0]]` is even less constrained. Thus, we detect the expression "`array[\result[0]][\result[1]]`" to not only potentially lead to a throw of an `IndexOutOfBoundsException`, but even a `NullPointerException`.

5.1.2. Undefined Method Invocation

This exception type does not exist in Java and was defined in terms of this work as to denote an invalid method invocation. The reason for this new exception type is that in JML/JML*, we can also invoke methods on specification level. However, as the ulterior motive is a modular one, such that we do not need to handle the Java code of a certain method after proving its method contracts, we do not want to consider a method's code when invoking it in a specification. Instead, we use the invoked method's method contracts. However, it can occur that when invoking a certain method m, its precondition is not fulfilled and thus no method contract for m can be used. In this case, we call the invocation to be *undefined* and imagine it to throw the exception `UndefinedMethodInvocation`.

5.1.2.1. Method in `CellClient`

```
class Cell {
    private int x;
    //@ accessible \inv: \nothing;
    ...
    /*@ model \locset footprint;
      @ accessible footprint: footprint;
      @ represents footprint = x;
      @*/

    /*@ normal_behavior
      @ assignable \nothing;
      @ accessible footprint;
      @ ensures \result == getX();
      @*/
    int getX() { ... }
}
class CellClient {
    //@ ensures \result.getX() == 5;
    Cell m() { ... }
}
```

Listing 5.6: Specifications for `CellClient.m()`

The class `CellClient` contains one method `m()` returning an object of type `Cell`. Furthermore, the class `Cell` for our purposes contains a global variable `x` of exact numerical value, a model field `footprint` representing the variable `x`, and a method `getX()` specified to not throw any exceptions, terminate normally and not have any side-effects, but accessing the model field `footprint`, returning an exact numerical value.
This specification in listing 5.6 is not *well-defined* for `getX()` is no helper method or constructor and thus for invoking it in a valid manner, the invariant of its receiver has to be guaranteed. As this is not the case for the specification of `m()`, the method invocation is *undefined*.

5.1.2.2. Method in `PrefixSumRec`

```
final class PrefixSumRec {
    ...
    /*@ normal_behavior
      @   requires x >= 0;
      @   ensures \result ==
      @       (\product int i; 0 <= i && i < x; 2);
      @   ensures \result > x;
      @   accessible \nothing;
      @   measured_by x;
      @ strictly_pure helper
      @*/
    private static int pow2( int x ) { ... }

    /*@ normal_behavior
      @   requires k >= 0;
      @   ensures 0 <= \result && \result <= k;
      @   ensures pow2(\result) <= k+1;
      @   ensures k % pow2(\result+1) == pow2(\result)-1;
      @   ensures (\forall int z; k% pow2(z+1) == pow2(z)-1;
      @               z >= \result);
      @   accessible \nothing;
      @ strictly_pure helper
      @*/
    private static int min ( int k ) { ... }
}
```

Listing 5.7: Specification for `PrefixSumRec.min(int)`

The class `PrefixSumRec` for our purposes contains the two static methods `pow2(int)` and `min(int)`, both specified to not throw any exceptions, terminate normally, not have any side-effects, not create any new objects and specified as helper methods. A helper method does not need to respect the invariant and also does not implicitly assume it

in the method's precondition. The method `pow2(int)` computes for an exact numerical value x greater or equal to zero the result of the function $f(x) = x^2$ and `min(int)` computes for an exact numerical value k greater or equal to zero the smallest exact numerical value n greater or equal to zero fulfilling "$n^2 \leq k+1 \wedge k \,\%\, (n+1)^2 = n^2 - 1$". The specification in listing 5.7 is not *well-defined* as $z+1$ and z do not fulfil the precondition of `pow2(int)` to be greater or equal to zero.

5.1.2.3. Method in `ExampleSubject`

```
public abstract class Subject {
    ...
    //@ public model \locset footprint;
    //@ accessible footprint: footprint;

}
public class ExampleSubject extends Subject {
    private int value;
    //@ represents footprint = value;
    ...
    /*@ normal_behaviour
      @   accessible footprint;
      @   ensures \result == value();
      @*/
    public /*@pure helper@*/ int value() { ... }
}
```

Listing 5.8: Specification for `ExampleSubject.value()`

The class `ExampleSubject` for our purposes contains a global variable `value` of exact numerical value, a model field `footprint` which represents this value and a method `value()`, which is specified as a helper method, to not throw any exceptions, to terminate normally and to not have any side-effects. Furthermore, `ExampleSubject` is a subclass of the abstract class `Subject`, from which it inherits the model field mentioned.
The specification in listing 5.8 is not *well-defined* as "`this.\inv`" has to hold for accessing the model field `this.footprint`.

5.1.2.4. Loop in `Client.listContainsMe(List)`

```
public interface List {
    //@ public instance invariant 0 <= size();
    //@ public accessible \inv: footprint;
    ...
    //@ public model instance \locset footprint;
    //@ public accessible footprint: footprint;
}
interface ListIterator {
    //@ accessible \inv: this.*, list.footprint;
    ...
    //@ public model instance List list;
    //@ accessible list: this.*;
    //@ public model instance int pos;
    //@ accessible pos: this.*;
}
public class Client {
    ...
    /*@ normal_behaviour
      @   requires list.\inv;
      @   ...
      @*/
    boolean /*@pure@*/ listContainsMe(List list) {
        ListIterator it = list.iterator();
        /*@ loop_invariant it.\inv && it.list == list
          @  && 0 <= it.pos && it.pos <= list.size()
          @  && (\forall int i; 0 <= i && i < it.pos;
          @                     list.get(i) != this);
          @ assignable it.*;
          @ decreases list.size() - it.pos;
          @*/
        while(...) { ... }
        ...
    }
}
```

Listing 5.9: Specification for `Client.listContainsMe(List)`

The class `Client` for our purposes contains the method `listContainsMe(List)`, which is specified to not throw any exceptions, terminate normally and not have any side-effects. This method contains one loop statement, on whose specification we will perform a *Well-Definedness Check*. It takes an object of type `List` as argument, for which we know that its invariant is guaranteed. The class `List` contains the model field `footprint`, on which its invariant depends. Furthermore, in the before mentioned method `listContainsMe(List)`

an object of type `ListIterator` is created through the element of type `List`. The class `ListIterator` contains the two model fields `list` of type `List` and `pos` of type `int` and its invariant depends on the model field `list.footprint`.
When checking the loop statement's *well-definedness* as shown in listing 5.9, we detect it to be not *well-defined* as `it.pos` in the variant (or decreases clause) might not be defined after a loop iteration as the loop is allowed to modify "`it.*`" (as stated in its *assignable*-clause).

5.1.2.5. Loop in `MySet.addAll(List)`

```
public class MySet {
    /*@ private invariant list.\inv && ...;
      @ public accessible \inv: footprint;
      @*/
    private List list;
    //@ private represents
    //@     footprint = this.*, list.footprint;
    ...
    /*@ normal_behaviour
      @   requires l.\inv;
      @   requires \disjoint(l.footprint, footprint);
      @   assignable footprint;
      @   ensures ...;
      @*/
    public void addAll(List l) {
        final ListIterator it = l.iterator();
        /*@ loop_invariant 0 <= it.pos && it.pos <= l.size()
          @   && (\forall int x; 0 <= x
          @                   && x < \old(list.size());
          @               list.get(x) == \old(list.get(x)));
          @ assignable l.footprint;
          @ decreases l.size() - it.pos;
          @*/
        while(...) { ... }
    }
}
```

Listing 5.10: Specification for `MySet.addAll(List)`

The class `MySet` for our purposes consists of one element `list` of type `List`, one model field `footprint` representing all fields of the class as well as the model field "`list.footprint`" and a method `addAll(List)` containing a loop statement. In the invariant of `MySet`, it is stated to depend on the model field `footprint` and the invariant for the element `list` is guaranteed. The classes `List` and `ListIterator` are as in listing 5.9.

When checking the *well-definedness* of the loop statement in listing 5.10, we detect it to be not *well-defined* as "`it.\inv`" and "`l.\inv`" are not guaranteed to hold after a loop iteration, because the loop is allowed to modify "`l.footprint`" (on which the invariant for `ListIterator` depends).

5.1.3. Null Pointer Exception

This exception type denotes the invalid use of a null reference. In practice, this means some field is referenced for the `null` object. That is perhaps one of the most common exception types in Java and often leads to problems in the course of writing and testing Java programs. It occurs usually, when we expect an object of a specific type, and invoke some method or refer to a specific field for it. However, when the element is the `null` object, this method or field is not defined and an exception is thrown.

5.1.3.1. Method in `LRS`

```
public class LRS {
    private int s = 0;
    private int t = 0;
    private int l = 0;
    private final SuffixArray sa;
    ...
    /*@ normal_behavior
      @ requires \invariant_for(sa) && sa.a.length >= 2;
      @ ensures 0 <= s && s < sa.a.length;
      @ ensures 0 <= t && t < sa.a.length;
      @ ensures 0 <= l && l < sa.a.length;
      @ ensures s+l <= sa.a.length && t+l <= sa.a.length;
      @ ensures (\forall int j; 0 <= j && j < l;
      @                         sa.a[s+j] == sa.a[t+j]);
      @ ensures s != t || l == 0;
      @ ensures !(\exists int i,k; 0 <= i && i < k
      @                         && k < sa.a.length-l;
      @            (\forall int j; 0 <= j && j <= l;
      @                            sa.a[k+j] == sa.a[i+j]));
      @*/
    public void doLRS() { ... }
}
```

Listing 5.11: Specification for `LRS.doLRS()`

The class `LRS` contains three variables of exact numerical value, one `SuffixArray` as in listing 5.4 and for our purposes one method `doLRS()` specified to not throw any exceptions and terminate normally, but potentially having side-effects. Its objective is to compute the longest repeated substring, and to store its length in the variable `l` and the

indices of the two suffixes with this prefix in the variables `s` and `t`. Therefore, the method needs the `SuffixArray` `sa` to respect its invariant and its array `a` to have a length of at least size two.
The specification in listing 5.11 is not *well-defined* as `sa.a` can potentially be `null` in the postcondition with `\everything` being the implicit *assignable*-clause. As a solution, we could either adjust the *assignable*-clause by only including the variables `s`, `t` and `l`, or prepend either a null-check for `sa.a` or the invariant for `sa` similar to the precondition.

5.1.3.2. Method in `ArrayList`

```
public class ArrayList implements List {
    ...
    /*@ private normal_behavior
      @   requires l >= 0;
      @   ensures \typeof(\result) == \type(Object[])
      @        && \result.length == l && \fresh(\result)
      @        && \result != null;
      @   assignable \nothing;
      @*/
    private /*@helper*/ /*@nullable*/Object[]
            newArray(int l) { ... }
}
```

Listing 5.12: Specification for `ArrayList.newArray(int)`

The class `ArrayList` for our purposes contains the private method `newArray(int)` and implements the interface `List` (which does not matter to the *well-definedness* of `newArray(int)` though). The specification of `newArray(int)` declares it to be a helper method, not throw any exceptions, terminate normally and to not have any side-effects. It returns an array of objects, which is specified to potentially be `null` though. For an exact numerical value `l` greater or equal to zero, the method returns a fresh array of size `l` not equal to zero (as stated in the postcondition).
The specification for `newArray(int)` in listing 5.12 is not *well-defined* as `\result` might be `null` in the event of accessing `\result.length`. However, this is only the case in the semantics of the $\mathcal{L}$-operator as at the end of the postcondition, it is asserted to not be equal to `null`.

5.1.3.3. Method in `Harness`

```
public class SparseArray {
    /*@ private invariant size == val.length
      @ && footprint == ...;
      @ public accessible \inv: size, footprint,
      @                         \singleton(footprint);
      @*/

    //@ public ghost int size;
    //@ public ghost \locset footprint;
    private int[] val, idx, back;
    private int n = 0;

    /*@ normal_behavior
      @   requires 0 <= i && i < size;
      @   accessible footprint;
      @   ensures \result == get(i);
      @*/
    public /*@pure@*/ int get(int i) { ... }
}
public class Harness {
    static /*@ nullable @*/ SparseArray a, b;
    ...
    /*@ normal_behaviour
      @   ensures a.get(5) == 0 && b.get(7) == 0;
      @*/
    static void sparseArrayTestHarness1() { ... }
}
```

Listing 5.13: Specification for `Harness.sparseArrayTestHarness1()`

The class `Harness` contains two static Objects `a` and `b` of type `SparseArray` specified nullable as well as a static method `sparseArrayTestHarness1()`. `SparseArray` contains the three arrays `val`, `idx` and `back` with elements of exact numerical value, one variable `n` of exact numerical value, one ghost variable `size` of exact numerical value, one ghost field `footprint` and a method `get(int)`, specified to not have any side-effects, to terminate normally and to not throw any exceptions. Variables and fields specified by the keyword `ghost` are elements, which do not play any role in the actual Java program, but are used to facilitate the specification. For means of specification, they can be used as if they were ordinary Java elements. As they cannot be changed by any Java instruction, JML/JML* offers so-called `set` statements, which are an equivalent to Java instructions on specification level, which cannot change ordinary Java elements though.
Regarding the *well-definedness* of `Harness.sparseArrayTestHarness1()` in listing 5.13, we detect an *ill-definedness*. This is, because `a` and `b` are nullable and not constrained

in any way as well. Thus, we get a `NullPointerException`, but also the invariants for either of them are not guaranteed to hold upon invoking `get(int)` as well as the method's precondition, requiring them to be greater or equal to zero and smaller than the ghost variable `size`. In our terms that would also be an `UndefinedMethodInvocation`.

5.1.4. Object Not Created Exception

This exception type does not exist in Java and was defined in terms of this work as to denote statements about objects which are potentially not created in the state of the statement made. It is similar to a `NullPointerException`, but also even worse in the sense that we might know the object is not equal to `null`, but we still cannot talk about it. Thus, we lack further knowledge about the object, i.e. it is not reachable.

5.1.4.1. Method in `ListOperationsNonNull`

```
public class ListNN {
    public ListNN next;
    public int value;
}
public class ListOperationsNonNull {
    /*@ public normal_behavior
      @ requires o != null && n >= 0
      @       && (\forall ListNN l; l.next != null);
      @ assignable \strictly_nothing;
      @ ensures  (n == 0 ==> \result == o)
      @       && (n >  0  ==> \result ==
      @             getNextContractNN(o,n-1).next);
      @ diverges true;
      @*/
    public ListNN getNextNN(ListNN o, int n) { ... }
    ...
    /*@ public normal_behavior
      @ requires  n>=0 && o!=null;
      @ assignable \strictly_nothing;
      @ accessible \infinite_union(ListNN l; l.next);
      @ ensures  (n == 0 ==> \result == o)
      @       && (n >  0  ==> \result ==
      @             getNextContractNN(o,n-1).next);
      @ measured_by n;
      @*/
    public ListNN getNextContractNN(ListNN o, int n) { ... }
}
```

Listing 5.14: Specification for `ListOperationsNonNull.getNextNN(ListNN,int)`

The class `ListOperationsNonNull` for our purposes contains the two methods `getNextNN(ListNN, int)` and `getNextContractNN(ListNN, int)`, both returning an object of type `ListNN`. This class `ListNN` contains the two fields `next` of type `ListNN` and `value` of type `int`. The method `getNextContractNN(ListNN, int)` is specified to not throw any exceptions, to terminate normally, not have any side-effects and to not create any new objects. And `getNextNN(ListNN, int)` is specified to not throw any exceptions, not have any side-effects and to not create any new objects. Both methods return some kind of n-th next element with respect to the given one.
We check the *well-definedness* for `getNextNN(ListNN,int)`'s specification in listing 5.14 and detect it to not be *well-defined* as `getNextContractNN(o, n-1)` can potentially be `null` and not created (hence throwing both a `NullPointerException` and an `ObjectNotCreatedException` as well) and thus an access to the field `next` might not be defined. A solution to this problem is to prepend the expression "`(\reach(o.next, o, getNextContractNN(o,n-1))) &&`" to the problematic reference in order to guarantee the object `getNextContractNN(o,n-1)` to be reachable via the Object `o` using its field `next`.

5.1.5. Class Cast Exception

This exception type denotes an invalid cast, usually a type cast. It means that a type cast to a type T is tried to be performed on some object o, although o cannot be cast to type T as it does not have a compatible type. That can occur for example when accessing elements in a data structure such as a list or a map, since the actual type of the contained element might not always be obvious. In this case the mentioned exception is thrown, since the cast is invalid, i.e. not *defined*.

5.1.5.1. Invariant in `SimplifiedLinkedList`

```
public final class Node {
    public /*@nullable@*/ Node next;
    public Object data;
}
final class SimplifiedLinkedList {
    /*@ private invariant
      @  (\forall int i; 0<=i && i<size;
      @          ((Node)nodeseq[i]) != null
      @       && (\forall int j; 0<=j && j<size;
      @             (Node)nodeseq[i] == (Node)nodeseq[j]
      @                ==> i == j)
      @          && ((Node)nodeseq[i]).next ==
      @             (i==size-1 ? null : (Node)nodeseq[i+1]))
      @ && size > 0 && first == (Node)nodeseq[0]
      @ && size == nodeseq.length;
      @*/
    private /*@nullable@*/ Node first;
    private int size;
    /*@ private ghost \seq nodeseq; */
    ...
}
```

Listing 5.15: Invariant specification for `SimplifiedLinkedList`

The class `SimplifiedLinkedList` implements (as the name implies) a simple version of an ordinary linked list. In order to achieve this, it contains an object `first` of type `Node` indicating the first element in the list, a variable `size` of exact numerical value storing the total amount of elements in the list, and a ghost variable `nodeseq` of sequence type. The object `first` is specified nullable and the invariant guarantees it to contain the first (with index 0) element of `nodeseq` as well as the fact that `size` is equal to the length of `nodeseq`. `Node` contains one nullable object `next` of type `Node` and one (non-nullable) object `data` of type `Object`.
The specification for the invariant in listing 5.15 is not *well-defined* as `nodeseq[i]` is cast to the type `Node` without knowing whether it actually is an object of this type.

Additionally (only for the semantics of the $\mathcal{L}$-operator), it is not known whether `i` is smaller than `nodeseq.length` or not (the guarding assertion is in the last line of the invariant), leaving us with a potential `IndexOutOfBoundsException`.

5.1.5.2. Method in `SimplifiedLinkedList`

```
final class SimplifiedLinkedList {
    ...
    /*@ normal_behaviour
      @ requires n >= 0 && n < size && \invariant_for(this);
      @ ensures \result == (Node)nodeseq[n];
      @ assignable \strictly_nothing;
      @ helper */
    private Node getNext(int n) { ... }
}
```

Listing 5.16: Specification for `SimplifiedLinkedList.getNext(int)`

The before mentioned (in listing 5.15) class `SimplifiedLinkedList` furthermore contains a private method `getNext(int)` returning an object of type `Node`. It is specified to be a helper-method, to not throw any exceptions, terminate normally, not have any side-effects and to not create any new objects. Its objective is to return the n-th element in the sequence `nodeseq` for n being an exact numerical value greater or equal to zero and smaller than `size`. As a further precondition, it requires the invariant of `SimplifiedLinkedList` to hold.
The specification in listing 5.16 would also potentially throw a `ClassCastException` and thus be not *well-defined* as we do not check the type of `nodeseq[n]` before casting it to an object of type `Node`. Here, it should be noted that the assertion `((Node) nodeseq[i]) != null` in `SimplifiedLinkedList`'s invariant is intended to guarantee a safe cast to `Node` for all elements of `nodeseq` and thus make the method contract for `getNext(int)` *well-defined*. However, here the general KeY calculus lacks a rule to deduce this.

5.1.5.3. Method in `Queens`

```
class Queens {
    ...
    /*@ public normal_behaviour
      @   requires 0 < n;
      @   ensures \result != null ==>
      @        \result.length == n
      @     && (\forall int x; 0 <= x && x < n;
      @            0 <= \result[x] && \result[x] < n)
      @         && (\forall int p, x;
      @                0 <= x && x < p && p < n;
      @                \result[x] != \result[p]
      @             && \result[x] - \result[p] != p - x
      @             && \result[p] - \result[x] != p - x);
      @   ensures \result == null ==>
      @        !(\exists \seq s; s.length == n
      @             && (\forall int x; 0 <= x && x < n;
      @                    0 <= (int)s[x] && (int)s[x] < n)
      @                 && (\forall int p, x;
      @                        0 <= x && x < p && p < n;
      @                        (int)s[x] != (int)s[p]
      @                     && (int)s[x] - (int)s[p]
      @                            != p - x
      @                     && (int)s[p] - (int)s[x]
      @                            != p - x));
      @*/
    public static /*@pure nullable@*/ int[] nQueens(int n) {
        ...
    }
}
```

Listing 5.17: Specification for `Queens.nQueens(int)`

The class `Queens` for our purposes contains one static method `nQueens(int)` returning an array of variables with exact numerical value. It is specified to not throw any exceptions, terminate normally and not have any side-effects. Furthermore, its result is specified to be nullable, i.e. to be potentially equal to `null`. Its objective is for any exact numerical value `n` greater than zero to compute an array of length `n` containing a solution to the popular *eight queens puzzle* with `n` instead of eight queens.
The specification in listing 5.17 is not *well-defined* as `s[x]` and `s[p]` are cast to variables of type `int` without knowing their type beforehand. As a solution, a simple prior type-check comes into mind.

5.1.5.4. Method in `Queens`

```
class Queens {
    ...
    /*@ private normal_behaviour
      @  requires 0 <= pos && pos < board.length;
      @   && (\forall int x; 0 <= x && x < board.length;
      @           0 <= board[x] && board[x] < board.length);
      @   && (\forall int p, x; 0 <= x && x < p && p < pos;
      @           board[x] != board[p]
      @        && board[x] - board[p] != p - x
      @        && board[p] - board[x] != p - x);
      @  ensures (\forall int x; 0 <= x && x < board.length;
      @            0 <= board[x] && board[x] < board.length);
      @   && \result ==> (\forall int p, x;
      @          0 <= x && x < p && p < board.length;
      @          board[x] != board[p]
      @       && board[x] - board[p] != p - x
      @       && board[p] - board[x] != p - x);
      @   && !\result ==> !(\exists \seq s;
      @          s.length == board.length;
      @          (\forall int x; 0 <= x && x < s.length;
      @              0 <= (int)s[x] && (int)s[x] < s.length)
      @       && (\forall int p, x;
      @              0 <= x && x < p && p < s.length;
      @              (int)s[x] != (int)s[p]
      @           && (int)s[x] - (int)s[p] != p - x
      @           && (int)s[p] - (int)s[x] != p - x)
      @       && (\forall int x; 0 <= x && x < pos;
      @              (int)s[x] == \old(board[x])));
      @  assignable board[pos..board.length-1];
      @  measured_by board.length - pos;
      @*/
    private static /*@nullable@*/ boolean
                    search(int pos, int[] board) { ... }
}
```

Listing 5.18: Specification for `Queens.search(int,int[])`

Again, we inspect the class `Queens`, but this time its static method `search(int,int[])` returning a boolean value. The method is specified to not throw any exceptions and terminate normally, potentially returning `null`. For an array `board` of values of exact numerical value between zero (including) and the array length (excluding) (each occurring once) and an exact numerical value `pos` inside the bounds of `board` with the first `pos` values in `boards` denoting a valid configuration for the *n queens puzzle*, the method

computes if there are conflicts (with respect to a valid solution to the *n queens puzzle*) in the whole array `board`.
This specification in listing 5.18 is again not *well-defined* as `s[x]` and `s[p]` are cast to `int` without knowing their type beforehand.

5.1.6. Negative Array Size Exception

This exception type denotes an array created with a negative size, however in our case we do not only relate this exception type to arrays, but also to similar data structures such as sequences, an abstract data type in JML*. A `NegativeArraySizeException` is less common than the exception types mentioned before, but it still needs to be handled. Usually, it does not occur when explicitly denoting the size of an array to be created, but when its size is computed by a complex function, which returns unexpected (negative) results in some cases. Arrays are only defined for a size bigger or equal to zero and thus a creation of one with negative size is not possible and *undefined*.

5.1.6.1. Method in `TestLists`

```
public interface List {
    //@ public instance invariant
    //@     \subset(\singleton(this.seq), footprint)
    //@  && \subset(\singleton(this.footprint), footprint)
    //@  && (\forall int i; 0<=i && i<seq.length;
    //@                     seq[i] instanceof Object);
    //@ public accessible \inv: footprint;
    ...
    //@ public ghost instance \locset footprint;
    //@ public ghost instance \seq seq;
}
public class TestLists {
    ...
    /*@ public normal_behaviour
      @  requires s >= 0 && s <= l.seq.length && t.\inv
      @   && l.\inv && \disjoint(t.footprint, l.footprint);
      @  assignable t.footprint;
      @  ensures \new_elems_fresh(t.footprint);
      @  ensures t.seq ==
      @   \seq_concat(\old(t.seq), \seq_sub(l.seq, 0, s-1));
      @  ensures t.\inv;
      @*/
    public static void append(List t, List l, int s) { ... }
}
```

Listing 5.19: Specification for `TestLists.append(List,List,int)`

The class `TestLists` for our purposes contains a static method `append(List,List,int)`. It is specified to not throw any exceptions and terminate normally. Furthermore, it gets parameters of type `List`, which is an interface with the two ghost fields `seq` of sequence type and `footprint` denoting a location set, on which its invariant depends. The location set denoted by `footprint` thereby contains `seq` and all fields in `List`. Finally, the inspected method takes two objects `t` and `l` of type `List`, requiring them to respect their invariants, and an exact numerical value `s` greater or equal to zero and less or equal to the size of `l`'s sequence of `l` in order to concatenate the first `s` elements of `l`'s sequence to `t`'s sequence, preserving `t`'s invariant.
The specification for the method `append(List,List,int)` in listing 5.19 is not *well-defined* as the size (`s-1`) of the new sequence is not guaranteed to be at least zero or bigger and potentially leads to a sequence creation of negative size. As such a creation is not defined, this leads to an exception. A simple solution to this problem would be to require `s` to be truly greater than zero.

5.1.6.2. Loop in `TestLists`

```
public class TestLists {
    ...
    /*@ public normal_behaviour
      @  ...
      @*/
    public static void append(List t, List l, int s) {
        int i = 0;
        /*@ loop_invariant
          @    0<=i && i<=s
          @ && t.seq == \seq_concat(\old(t.seq),
          @                         \seq_sub(l.seq, 0, i-1))
          @ && \new_elems_fresh(t.footprint)
          @ && \disjoint(t.footprint, l.footprint)
          @ && \invariant_for(t) && \invariant_for(l);
          @ assignable t.footprint;
          @ decreases s - i;
          @*/
        while (...) { ... }
    }
}
```

Listing 5.20: Loop specification in `TestLists.append(List,List,int)`

Again, we refer to the before mentioned method `append(List,List,int)` from listing 5.19, but now have a look at the loop statement (listing 5.20) in it. Therein, we also use the parameters `t`, `l` and `s` as well as their (`t`'s and `l`'s) ghost fields `footprint` and `seq`, but also a local variable `i` of exact numerical value.

This specification in listing 5.20 is not *well-defined* as the size (`i-1`) of the new sequence resulting from the concatenation is not guaranteed to be at least zero or bigger. As a solution, we could either start with the value 1 instead of 0 and also adjust the loop invariant likewise or define this special case differently in the specification.

5.1.6.3. Method in `LinkedList`

```
class LinkedList {
    /*@ public invariant \subset(this.*, repr);
      @ public accessible \inv: repr
      @ \measured_by seq.length;
      @ private invariant length == seq.length;
      @   && length != 0 ==> seq[0] == head;
      @   && tail == null ==> length <= 1;
      @   && tail != null ==> 2 <= length
      @                    && \subset(tail.*, repr)
      @                    && \subset(tail.repr, repr)
      @                    && \disjoint(this.*, tail.repr)
      @                    && seq[1..length] == tail.seq
      @                    && tail.\inv;
      @*/
    private final int head;
    private final /*@nullable@*/ LinkedList tail;
    private final int length;
    //@ public final ghost \seq seq;
    //@ public final ghost \locset repr;
    ...
    /*@ public normal_behaviour
      @   ensures \result.\inv;
      @   ensures \result.seq == seq[1..seq.length];
      @   ensures \fresh(\set_minus(\result.repr, repr));
      @*/
    public /*@pure@*/ LinkedList tail() { ... }
}
```

Listing 5.21: Specification in `LinkedList.tail()`

Let us now have a look at the class `LinkedList` denoting yet another implementation for the well-known data structure of the same name. It contains the two variables `head` and `length` of exact numerical value, an object `tail` of type `LinkedList` specified nullable as well as the ghost object `seq` of sequence type and the ghost field `repr` denoting a location set. Furthermore, the length of `seq` forms the *measured_by*-value for the class, guaranteed by the invariant to be equal to the variable `length`. For our purposes, we inspect the method `tail()` contained in the class mentioned. This method returns an

object (specified non-nullable) of type `LinkedList` and is specified to not throw any exceptions, terminate normally and not have any side-effects. In its postcondition, it also guarantees the invariant for the result object to hold.
The specification for `tail()` in listing 5.21 turns out to be not *well-defined* as the new sequence `seq` of `\result` is not guaranteed to have a size bigger than zero. This is, because the length of `seq` is not necessarily truly greater than zero. If it is equal to zero, the sub sequence created by `seq[1..seq.length]` would have a negative size (equal to -1), thus making the creation *undefined*. Possible solutions as prior described for `tail()` come to mind.

5.1.7. Arithmetic Exception

This exception type denotes an arithmetic error, such as divide-by-zero. It occurs for basic mathematical functions such as division or modulo, which are not *defined* for all possible parameters. An `ArithmeticException` is directly related to fundamental mathematical concepts and can as such also occur in many other modern programming languages. Hence, it is the most fundamental or classical exception type and needs to be dealt with for any expression denoting numerical operations.

5.1.7.1. Method in `GreatestCommonDivisor`

```
public class GreatestCommonDivisor {
    ...
    /*@ public normal_behavior
      @ requires a != 0 || b != 0;
      @ ensures (a % \result == 0 && b % \result == 0 &&
      @          (\forall int x; x > 0 && a % x == 0
      @                && b % x == 0; \result % x == 0));
      @ assignable \nothing;
      @*/
    public static int ofWith(final int a, final int b) { ...
    }
}
```

Listing 5.22: Method specification in `GreatestCommonDivisor`

Finally, we have a look at the class `GreatestCommonDivisor` for our purposes containing the static method `ofWith(in, int)`. The method is specified to not throw any exceptions, terminate normally and not have any side-effects. Furthermore, it requires at least one of their arguments `a` and `b` of exact numerical value to be non-zero and computes their greatest common divisor.
The method's specification in listing 5.22 is not *well-defined* as `\result` can potentially be equal to zero and thus e.g. the expression "`a % \result`" would be *undefined*. This would lead to an `ArithmeticException` as the modulo operation is *undefined* for the second argument being equal to zero.

5.2. Performance

Until now, the case study only considered one of the two semantics presented and implemented within this thesis, namely the $\mathcal{L}$-operator semantics. As we justified this choice in the preface with a difference in performance, this claim still needs validation. This section will catch up on this by presenting some statistics illustrating the performance difference for *Well-Definedness Checks* performed within this case study. Unlike the last section, this will only cover *Well-Definedness Checks* for specifications found to be *well-defined*, thereof we examined the performance for a set of method specifications in the KeY examples – a set denoted by the name `firstTouch` and meant to be a means for new KeY users to get to know its main functionalities – and used the automatic proof mechanism in KeY to check their *well-definedness*.
For the measurement results, whose tables can be found in appendix A, several different measures are given and presented. They are grouped by their according example package. Each result is denoted by the method name and (if multiple contracts are specified for it) the specified behaviour with an index number, forming the contract name. Furthermore, we give the size of the explicit specification (lines of specification or LoS), measures describing the size of the proof tree (amount of nodes and branches), the total time needed to automatically prove the *well-definedness*, the average time per proof step or node of the proof tree, the amount of aggregated simplification rules (performed by the KeY *One Step Simplifier* or OSS) and finally the total amount of rule applications (apps). Each *Well-Definedness Check* is performed and measured for both the $\mathcal{L}$- and the $\mathcal{Y}$-operator, each having a different semantics. Although we also did a direct implementation for the $\mathcal{D}$-operator, we did not perform checks with this operator in the case study at all. This was simply, because checks with it were not feasible due to the exponential growth of the size of the generated formulas. Even *Well-Definedness Checks* for very small method contracts took very long or needed more memory than there was available on the machine (two cores with 2.66 GHz each and 4 GB of RAM), on which we performed the proofs. In the following, we clarify our results by correlating our measurements for both operators with each other.

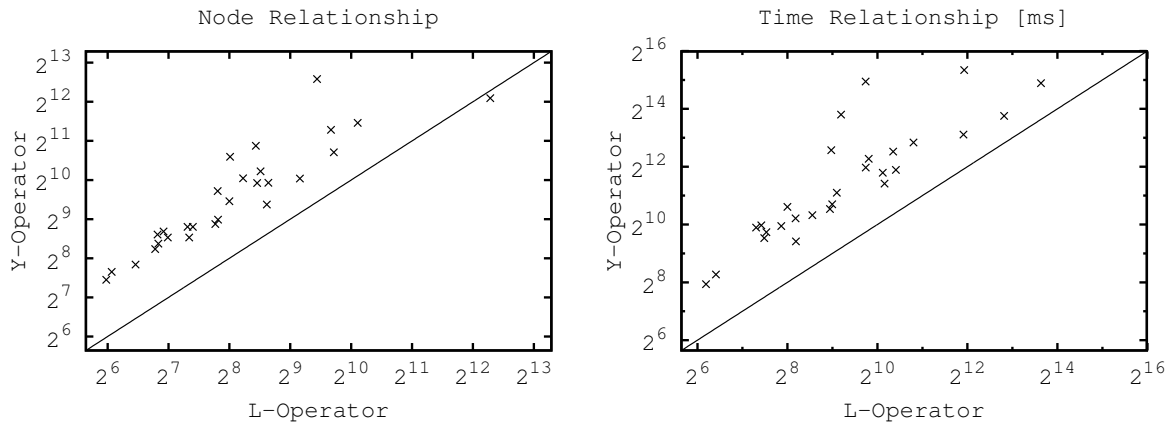

Figure 5.1.: Relationship of proof nodes and time between $\mathcal{L}$- and $\mathcal{Y}$-operator semantics

In fig. 5.1, the different performance results for the above mentioned examples with both operators are depicted. There, it can be seen, that – although both generate conditions which are linear in size with respect to their input formula – the $\mathcal{L}$-operator clearly beats the $\mathcal{Y}$-operator in terms of the proof size as well as the time needed to perform the automatic proof in KeY. However, these measurements are based on a rather small set, i.e. 28, of method contracts, which are all rather small in size (11 lines of specification at most). And moreover, compared to the actual proof obligations, the performance difference is not very significant.
Figure 5.1 also illustrates the two operators' performance relationship to each other. The mean difference is not very big and the numbers for the $\mathcal{Y}$-operator are about five times bigger than the numbers for the $\mathcal{L}$-operator. This is except for three method contracts. We suspect the reason for one of them (`SITA3.rearrange()`) to be the occurrence of nested universal quantifiers, which necessitate considerable more quantifier instantiations. For the other deviations, causes are not as clearly identifiable. Supplementary to the structure of the specification, the KeY calculus heuristics (i.e. the decision on when to apply which rule on which expression) could also play a crucial role in the difference.

6. Conclusion

A common approach to deal with the potential *undefinedness* of specification expressions is the automatic generation of *well-definedness* conditions. Proving these conditions to be fulfilled guarantees that no *undefined* expressions in specifications influence the evaluation process. At the very latest with the decision of the JML community of changing their notion of validity to specifically consider *undefined* expressions invalid, it became inevitable to tackle the issue of ensuring *well-definedness* of JML specifications.

6.1. Results

Within this work, we realised fully functional *Well-Definedness Checks* for JML/JML* specifications in the KeY System. Our procedure has been implemented in the KeY System to enforce *well-definedness* of class invariants, model fields, method contracts, loop statements and block contracts. *Well-Definedness Checks* form a preliminary step before the ordinary proof and reject *ill-defined* specifications. The thesis furthermore includes definitions for their ruleset, incorporating rules handling JML* expressions as well as rules handling complete specification elements (the ones mentioned above). Checks for class invariants, model fields and method contracts are implemented as separate proof obligations considering only the specification, whereas checks for loop statements and block contracts are designed as "on-the-fly" checks during the ordinary program verification, since they can only be seen in context of the state in which they are specified. Furthermore, we employed and extended two different semantics of *well-definedness*, the $\mathcal{L}$-operator semantics and the $\mathcal{D}$-operator semantics, and integrated efficient implementations in the KeY System. One evaluates based on classical logic, whereas the other simulates a *run-time assertion check* (RAC). Regarding the implementation, we extended the KeY System by adding short-circuit operators, making necessary arrangements to adjust the order in specification clauses and implementing new operators, namely *term transformers*, for the *Well-Definedness Operators*. Subsequently, we performed an extensive case study on the *well-definedness* of a major part of the KeY Examples and discussed problematic, i.e. *ill-defined*, specifications. We based our discussions on the types of `Java Unchecked RuntimeException` and also discovered new *undefinedness* problems, for which we introduced two new exception types to cope with JML/JML*'s richer expressiveness. Finally, we analysed the performance of the implemented *Well-Definedness Checks*, compared the two different semantics mentioned above, and illustrated our measurements.

6.2. Outlook and Future Work

The implemented *Well-Definedness Checks* can be used from now on in the KeY System and their practical gain and further benefits are yet to determine. More specifically, we could imagine an embeddeding in a more extensive *well-formedness* check, which also checks the satisfiability and *well-foundedness* (meaning there are no infinite descending chains in recursive specifications).
Based on this work, we also identified two particular tasks as useful extensions to our checks. Firstly, they could be extended to cover more JML/JML* specification elements as e.g. history constraints and set statements. Whereas most of them are presumably simple extensions to the implemented checks, some require more precise definitions as to be able to talk about their *well-definedness*.
Secondly, *well-definedness* does not only apply to the specification, but also to the verification process. Calculus rules, e.g. the *cut*-, *forall-left*- and *exists-right*-rule, can introduce potentially *ill-defined* expressions as well. Hence, a *Well-Definedness Check* for newly introduced expressions would also be a useful extension.
Finally, a further examination to evaluate the practical use of our implementation as part of the ordinary program verification process comes to mind. This requires an exploration over time to fully tackle the impact and implications on the KeY System and further evolutions on the field.

A. Performance Measurements

A.1. `SumAndMax`

Class `SumAndMax`

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
sumAndMax(int[])	8	$\mathcal{L}$	841 / 6	7208 / 8.5 (ms)	125	1252
		$\mathcal{Y}$	1674 / 7	13843 / 8.2 (ms)	297	2459

Table A.1.: Performance Measurements for `SumAndMax`

A.2. `LinkedList`

Class `Node`

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
cons(int,Node)	6	$\mathcal{L}$	299 / 8	856 / 2.8 (ms)	87	593
		$\mathcal{Y}$	1056 / 10	4011 / 3.7 (ms)	282	2038
search()	6	$\mathcal{L}$	365 / 5	1112 / 3.0 (ms)	77	659
		$\mathcal{Y}$	1194 / 10	3541 / 2.9 (ms)	239	1779

Table A.2.: Performance Measurements for `Node`

A.3. `Cell`

Class `Cell`

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
Cell()	4	$\mathcal{L}$	88 / 1	292 / 3.3 (ms)	33	200
		$\mathcal{Y}$	229 / 1	683 / 2.9 (ms)	76	476
getX()	4	$\mathcal{L}$	110 / 2	185 / 1.6 (ms)	30	189
		$\mathcal{Y}$	301 / 1	853 / 2.8 (ms)	80	491
setX(int)	4	$\mathcal{L}$	162 / 2	376 / 2.3 (ms)	44	295
		$\mathcal{Y}$	370 / 1	1280 / 3.4 (ms)	103	705

Table A.3.: Performance Measurements for `Cell`

A.4. `Java5`

Class `For`

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
infiniteLoop()	3	$\mathcal{L}$	159 / 3	256 / 1.6 (ms)	38	206
		$\mathcal{Y}$	447 / 4	1562 / 3.4 (ms)	91	592
sum()	1	$\mathcal{L}$	256 / 2	548 / 2.1 (ms)	60	460
		$\mathcal{Y}$	703 / 3	2198 / 3.1 (ms)	156	1208

Table A.4.: Performance Measurements for `For`

A.5. Quicktour

Class CardException

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
getCause()	2	$\mathcal{L}$	127 / 2	233 / 1.8 (ms)	39	230
		$\mathcal{Y}$	370 / 1	987 / 2.6 (ms)	105	616
getMessage()	3	$\mathcal{L}$	114 / 2	179 / 1.5 (ms)	30	143
		$\mathcal{Y}$	332 / 1	739 / 2.2 (ms)	83	432
initCause(Throwable)	4	$\mathcal{L}$	224 / 4	503 / 2.2 (ms)	62	343
		$\mathcal{Y}$	843 / 1	6087 / 7.2 (ms)	189	1300

Table A.5.: Performance Measurements for CardException

Class LogFile

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
LogFile()	6	$\mathcal{L}$	571 / 4	1361 / 2.3 (ms)	102	1045
		$\mathcal{Y}$	1051 / 5	3792 / 3.6 (ms)	244	1805
addRecord(int)	11	$\mathcal{L}$	4987 / 78	12731 / 2.5 (ms)	607	7433
		$\mathcal{Y}$	4367 / 30	30325 / 6.9 (ms)	705	7211
getMaximumRecord()	7	$\mathcal{L}$	814 / 10	1786 / 2.1 (ms)	124	1175
		$\mathcal{Y}$	2491 / 22	7310 / 2.9 (ms)	435	3662

Table A.6.: Performance Measurements for LogFile

Class LogRecord

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
getBalance()	3	$\mathcal{L}$	63 / 1	73 / 1.1 (ms)	20	75
		$\mathcal{Y}$	175 / 1	245 / 1.4 (ms)	45	210
getTransactionId()	3	$\mathcal{L}$	67 / 1	85 / 1.2 (ms)	23	102
		$\mathcal{Y}$	201 / 1	309 / 1.5 (ms)	57	303
setRecord(int)	7	$\mathcal{L}$	225 / 2	510 / 2.2 (ms)	65	428
		$\mathcal{Y}$	507 / 1	1665 / 3.2 (ms)	137	897

Table A.7.: Performance Measurements for LogRecord

Class `PayCard`

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
PayCard()	6	$\mathcal{L}$	218 / 2	493 / 2.2 (ms)	70	445
		$\mathcal{Y}$	470 / 1	1486 / 3.1 (ms)	146	881
PayCard(int)	8	$\mathcal{L}$	392 / 3	1143 / 2.9 (ms)	85	692
		$\mathcal{Y}$	664 / 1	2730 / 4.1 (ms)	176	1193
charge(int).exceptional	3	$\mathcal{L}$	121 / 2	158 / 1.3 (ms)	33	159
		$\mathcal{Y}$	412 / 2	951 / 2.3 (ms)	85	556
charge(int).normal.0	7	$\mathcal{L}$	351 / 6	896 / 2.5 (ms)	79	553
		$\mathcal{Y}$	972 / 1	4945 / 5.0 (ms)	206	1522
charge(int).normal.1	7	$\mathcal{L}$	346 / 6	854 / 2.4 (ms)	83	546
		$\mathcal{Y}$	1879 / 9	31532 / 16.7 (ms)	313	3030
chargeAndRecord(int)	5	$\mathcal{L}$	258 / 3	585 / 2.2 (ms)	56	370
		$\mathcal{Y}$	1544 / 4	14283 / 9.2 (ms)	242	2153
createJuniorCard()	6	$\mathcal{L}$	169 / 1	290 / 1.7 (ms)	72	457
		$\mathcal{Y}$	447 / 1	1188 / 2.6 (ms)	167	928
isValid()	5	$\mathcal{L}$	113 / 2	171 / 1.5 (ms)	35	202
		$\mathcal{Y}$	390 / 1	1003 / 2.5 (ms)	108	687

Table A.8.: Performance Measurements for `PayCard`

A.6. `SITA`

Class `SITA3`

Contract	LoS	Operator	Nodes / Branches	Time / Per Step	OSS	Apps
commonEntry(int,int)	9	$\mathcal{L}$	1103 / 19	3858 / 3.4 (ms)	134	1671
		$\mathcal{Y}$	2821 / 34	8846 / 3.1 (ms)	358	4227
rearrange()	6	$\mathcal{L}$	694 / 6	3898 / 5.6 (ms)	135	1262
		$\mathcal{Y}$	6126 / 45	41586 / 6.7 (ms)	908	8397
swap(int[],int,int)	6	$\mathcal{L}$	399 / 4	1306 / 3.2 (ms)	71	605
		$\mathcal{Y}$	976 / 1	5888 / 6.0 (ms)	166	1550

Table A.9.: Performance Measurements for `SITA3`

References

[AM02] Jean-Raymond Abrial and Louis Mussat. "On Using Conditional Definitions in Formal Theories". In: *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*. ZB '02. London, UK, UK: Springer-Verlag, 2002, pp. 242–269. ISBN: 3-540-43166-7. URL: http://dl.acm.org/citation.cfm?id=647285.723108 (cit. on pp. 2, 3).

[BBM98] Patrick Behm, Lilian Burdy, and Jean-Marc Meynadier. "Well Defined B". In: *Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*. B '98. London, UK, UK: Springer-Verlag, 1998, pp. 29–45. ISBN: 3-540-64405-9. URL: http://dl.acm.org/citation.cfm?id=647418.725483 (cit. on pp. 3, 12, 23).

[BCJ84] Howard Barringer, J. H. Cheng, and Cliff B. Jones. "A Logic Covering Undefinedness in Program Proofs". In: *Acta Inf.* 21 (1984), pp. 251–269 (cit. on p. 2).

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 3-540-68977-X, 978-3-540-68977-5.

[BKS07] Bernhard Beckert, Vladimir Klebanov, and Steffen Schlager. "Dynamic Logic". In: Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach*. Berlin, Heidelberg: Springer-Verlag, 2007. Chap. 3, pp. 69–177. ISBN: 3-540-68977-X, 978-3-540-68977-5 (cit. on p. 10).

[Bru09] Daniel Bruns. "Formal Semantics for the Java Modeling Language". Diploma thesis. Universität Karlsruhe, June 2009. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012399 (cit. on p. 11).

[Cha05] Patrice Chalin. *Logical Foundations of Program Assertions: What do Practitioners Want?* Montréal, Québec, Canada: Department of Computer Science and Software Engineering, Concordia University, 2005. URL: http://users.encs.concordia.ca/~chalin/papers/TR-2005-002-r2.pdf (cit. on pp. 4, 12).

[Dij75] Edsger W. Dijkstra. "Guarded commands, non-determinancy and a calculus for the derivation of programs". In: *Language Hierarchies and Interfaces*. Ed. by Friedrich L. Bauer and Klaus Samelson. Vol. 46. Lecture Notes in Computer Science. Springer, 1975, pp. 111–124. ISBN: 3-540-07994-7 (cit. on pp. 11, 36).

[DMR08] Ádám Darvas, Farhad Mehta, and Arsenii Rudich. *Efficient Well-Definedness Checking*. Zürich, Switzerland: ETH Zürich, 2008, pp. 100–115. URL: http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=DarvasMehtaRudich08.pdf (cit. on pp. 3, 13, 24).

[Gie04] Martin Giese. "Taclets and the KeY Prover". In: *Electr. Notes Theor. Comput. Sci.* 103 (2004), pp. 67–79 (cit. on p. 35).

[Häh05] Reiner Hähnle. "Many-Valued Logic, Partiality, and Abstraction in Formal Specification Languages." In: *Logic Journal of the IGPL* 13.4 (2005), pp. 415–433. URL: http://dblp.uni-trier.de/db/journals/igpl/igpl13.html#Hahnle05 (cit. on p. 2).

[Hol91] Marit Holden. "Weak logic theory". In: *Theoretical Computer Science* 79.2 (1991), pp. 295 –321. ISSN: 0304-3975. DOI: http://dx.doi.org/10.1016/0304-3975(91)90334-X. URL: http://www.sciencedirect.com/science/article/pii/030439759190334X (cit. on p. 2).

[Kas06] Ioannis T. Kassios. "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions". In: *FM*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 268–283. ISBN: 3-540-37215-6 (cit. on p. 8).

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "Preliminary Design of JML: A Behavioral Interface Specification Language for Java". In: *ACM SIGSOFT Software Engineering Notes* 31.3 (Mar. 2006), pp. 1–38. URL: http://doi.acm.org/10.1145/1127878.1127884 (cit. on p. 7).

[LC06] Gary T. Leavens and Yoonsik Cheon. *Design by Contract with JML*. Sept. 2006 (cit. on p. 7).

[Lea+11] Gary T. Leavens et al. *JML Reference Manual*. Available from http://www.jmlspecs.org. July 2011 (cit. on pp. 4, 7, 16, 19).

[RDM08] Arsenii Rudich, Ádám Darvas, and Peter Müller. "Checking Well-Formedness of Pure-Method Specifications". In: *Formal Methods (FM)*. Ed. by J. Cuellar and T. Maibaum. Vol. 5014. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 68–83 (cit. on p. 11).

[RL05] Arun D. Raghavan and Gary T. Leavens. *Desugaring JML Method Specifications*. 00-03e. Iowa State University, Department of Computer Science, May 2005. URL: ftp://ftp.cs.iastate.edu/pub/techreports/TR00-03/TR.pdf (cit. on p. 8).

[Rüm07] Philipp Rümmer. "Construction of Proofs". In: Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach*. Berlin, Heidelberg: Springer-Verlag, 2007. Chap. 4, pp. 179–242. ISBN: 3-540-68977-X, 978-3-540-68977-5 (cit. on p. 36).

[Sch11] Peter H. Schmitt. *TA Computer-Assisted Proof of the Bellman-Ford Lemma*. Karlsruhe, Mar. 2011. URL: `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022513` (cit. on pp. 2, 3).

[Sup57] P. Suppes. "Theory of Definition". In: *Introduction to logic*. New York: D. Van Nostrand Company, 1957. Chap. 8, pp. 151–174 (cit. on p. 2).

[Ulb08] Mattias Ulbrich. *A Set-Theoretic Model for Java States*. Unpublished. Karlsruhe, Germany: Universität Karlsruhe, 2008 (cit. on p. 14).

[Ulb09] Mattias Ulbrich. *Welldefinedness of Method Contracts in JML*. Unpublished. Karlsruhe, Germany: Universität Karlsruhe, July 2009 (cit. on pp. 4, 11, 14).

[Wac12] Simon Wacker. "Blockverträge". German. Studienarbeit. Karlsruhe Institute of Technology, Oct. 2012 (cit. on pp. 33, 34).

[Wei11] Benjamin Weiß. "Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction". PhD thesis. Karlsruhe Institute of Technology, 2011 (cit. on p. 8).