Karlsruhe Institute of Technology

A Divide-and-Conquer Strategy with Block and Loop Contracts for Deductive Program Verification

Bachelor's Thesis of

Florian Lanzinger

at the Department of Informatics
Institute of Theoretical Informatics (ITI)

Reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Dipl.-Inform. Michael Kirsten
Second advisor:	Dr. rer. nat. Mattias Ulbrich

15 December 2017 – 14 April 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

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, and that I have observed the KIT's rules for good scientific practice (*KIT-Satzung zur Sicherung guter wissenschaftlicher Praxis*).

Karlsruhe, 14 April 2018

. . . .

(Florian Lanzinger)

Abstract

Deductive program verification allows programmers to prove that their program behaves correctly for every valid input. For this, the programmer usually declares a contract for every method; this contract specifies the method's preconditions, i.e., for which inputs it will behave correctly, and its postconditions, i.e., what conditions its output must fulfill.

Functional correctness proofs generated from such specifications quickly become very long and complex as every possible branch in a method's execution leads to a branch in the proof. One way of dealing with this complexity is to divide a long method into shorter sub-methods which are easier to verify. When a method calls another method, we only prove that the callee's precondition is valid and then assume its postcondition without considering the callee's code. The callee's correctness is proven separately. However, the refactoring required by this approach is very time-consuming. The approach is further diminished by the fact that such a refactored version will in most cases be unsuited for production as the division of every method into easy to verify sub-methods leads to an unoptimized and convoluted program.

In this thesis, we introduce a new rule for JavaDL that allows the user to divide a method into blocks whose correctness may be proven independently of the method's correctness. For this, we use block contracts, a concept introduced by [Wac12] that allows the user to specify a contract for any block inside of a method. Using our new rule, these blocks can be applied like methods, i.e., we show that a block's precondition is valid at the point at which it occurs, and then assume its postcondition without considering the code inside the block.

As the verification of unbounded loops requires special treatment (because the unwinding of an unbounded loop does not terminate), we introduce a second rule for the application of blocks that start with a loop.

Instead of using loop invariants for this, we use the concept of loop specifications from [Tue12]. This allows us to specify loops using a kind of specialized block contract with pre- and postconditions instead of loop invariants.

Zusammenfassung

Deduktive Programmverifikation erlaubt es Programmierern, zu beweisen, dass ihr Programm sich für jede gültige Eingabe richtig verhält. Hierfür gibt der Programmierer in der Regel für jede Methode einen Vertrag an; dieser Vertrag spezifiziert die Vorbedingungen der Methode, d.h. für welche Eingaben die Methode sich korrekt verhält, sowie ihre Nachbedingungen, d.h. die Bedingungen, die die Ausgabe der Methode erfüllen muss.

Funktionale Korrektheitsbeweise, die aus solchen Spezifikationen generiert werden, werden schnell sehr lang und komplex, da jede mögliche Verzweigung in der Ausführung der Methode auch zu einer Verzweigung im Beweis führt. Eine Möglichkeit, mit dieser Komplexität umzugehen, ist, eine lange Methode in kürzere Untermethoden zu teilen, welche einfacher zu verifizieren sind. Wenn eine Methode eine andere Methode aufruft, beweisen wir nur, dass die Vorbedingung der aufgerufenen Methode wahr ist und nehmen dann ihre Nachbedingung an, ohne den Code der aufgerufenen Methode zu beachten. Die Korrektheit der aufgerufenen Methode wird separat bewiesen. Allerdings ist die Refaktorisierung, die diese Vorgehensweise voraussetzt, sehr zeitaufwändig. Die Vorgehensweise wird außerdem durch die Tatsache verschlechtert, dass eine solche refaktorisierte Version in den meisten Fällen nicht für den Einsatz im finalen Produkt geeignet sein wird, da die Aufteilung jeder Methode in einfach zu verifizierende Untermethoden ein unoptimiertes und kompliziertes Programm zur Folge hat.

In dieser Arbeit stellen wir eine neue Regel für JavaDL vor, die es dem Benutzer erlaubt, eine Methode in Blöcke zu unterteilen, deren Korrektheit unabhängig von der umgebenden Methode bewiesen werden kann. Hierfür nutzen wir Blockverträge, ein Konzept, das in [Wac12] eingeführt wurde und es dem Benutzer erlaubt, für jeden beliebigen Block innerhalb einer Methode einen Vertrag anzugeben. Mit unserer neuen Regel können diese Blöcke wie Methoden angewendet werden, d.h. wir zeigen, dass die Vorbedingung des Blocks an der Stelle, an der dieser auftritt, gültig ist, und nehmen dann seine Nachbedingung an, ohne den Code im Block zu beachten.

Da die Verifikation unbeschränkter Schleifen gesondert behandelt werden muss (weil die Ausrollung einer unbeschränkten Schleife nicht terminiert), stellen wir außerdem eine zweite Regel zur Anwendung von Blöcken, die mit einer Schleife beginnen, vor.

Statt hierfür Schleifeninvarianten zu nutzen, nutzen wir das Konzept von Schleifenspezifikationen aus [Tue12]. Dies erlaubt es uns, Schleifen mit einer Art spezialisiertem Blockvertrag mit Vor- und Nachbedingungen statt mit Schleifeninvarianten zu spezifizieren.

Contents

1. Introduction

As programs become bigger and more complex, being able to verify their reliability becomes ever more important.

The usual way this is achieved is by testing the program – or, more specifically, a single method within the program – with certain inputs and verifying the outputs. However, testing cannot show that a method is correct for *every* possible input.

In safety-critical applications especially, we may want to prove that a method satisfies certain properties for every input.

Deductive program verification refers to the practice of proving these properties by applying a logical calculus.

Proofs that involve complex or long methods can easily become quite difficult to follow, as every branch in the program's execution (e.g. *if* statements, or accesses to nullable object variables) leads to a branch in the proof.

One way of dealing with this complexity is to divide the method into sub-methods, turning local variables needed by more than one sub-method into attributes. For example, this approach was used in [BSSU17] to prove the correctness of a dual-pivot quicksort algorithm in Java's standard library.

However, this approach is very time-consuming, as it requires large-scale refactoring. Furthermore, the refactored version is often not useful for anything other than verification, as it is unoptimized and the transformation of local variables into attributes makes the code somewhat convoluted. Furthermore, if these new attributes are not reset to their original values after every method call (which would make the refactored program not equivalent to the original one), the program becomes non-reentrant.

In this thesis, we present two new rules for a sequent calculus for *JavaDL*, a dynamic logic for Java (a *dynamic logic* being a modal logic whose formulas may contain, and reason about, program fragments). The purpose of these two rules is to divide an existing proof into two sub-proofs which can then be proven independently of each other.

1.1. Tools and Techniques

The program specifications considered in this thesis are not written directly in JavaDL, but in a dialect of *JML* (*Java Modeling Language*) [LPC+13], a behavioral interface specification language for Java.

JML allows Java programmers to use the *Design by Contract* methodology by providing behavioral specifications of methods, loop, and blocks.

The behavior of a method consists of its precondition, which describes the program states for which the method is defined, one postcondition for every possible type of termination, a diverges condition, which describes the states in which the method may not terminate at all, and a frame, which describes which heap locations the method may change [LPC+13, 1.1].

Typically, a method's behavior is viewed as a contract between itself, the callee, and its caller: The caller must ensure that the callee's precondition is valid. In return, the callee guarantees its postcondition.

For the translation from Java and JML to JavaDL, we will use a program called *KeY* [ABB+16], which is is a system for the deductive verification of Java programs.

KeY translates a given program, along with its JML specifications, into several proof obligations. Every proof obligation is a sequent of JavaDL formulas, which is valid if and only if the program satisfies the specified properties.

When the user has selected a proof obligation, KeY tries to prove its validity by repeatedly applying rules from the sequent calculus.

1.2. Goals of this Thesis

The goal of this bachelor thesis is to provide a way to divide proofs into smaller, independent subproofs without having to perform large-scale refactoring. This will be achieved by using block contracts, a JML extension introduced in [Wac12]. Block contracts allow the programmer to divide the method to be verified into code blocks and specify a contract for every block.

[Wac12] uses three premisses to prove the method contract of a method containing a block:

1. *Validity:* The block contract is valid.
2. *Precondition:* Before the block is executed, its precondition is valid.
3. *Usage:* If the block's postcondition is valid, then, after the rest of the surrounding method has been executed, the method's postcondition is valid.

All three branches, including the validity branch, depend on the context in which the block occurs.

This has some advantages. First and foremost, it allows block contracts to not be universally valid. This makes sense intuitively: Unlike a method, which can be called from any context and thus should have a universally valid contract, a block only occurs once in the whole program, and thus its contract only needs to be valid in the one context in which it occurs.

However, instead of dividing the proof into shorter, independent sub-proofs, this rule only divides it into three branches, all of which are dependent on the original proof.

In chapter 4, we will introduce a modified rule which allows the user to prove the validity branch separately, outside of the surrounding method's context.

Loop invariants are another form of auxiliary specification, allowing us to specify a part of a larger method, in this case a loop [HAGH16, 9.2].

Currently, KeY uses three premisses to prove the method contract of a method containing a loop:

1. *Invariant initially valid:* The loop invariant is valid before the loop is first entered.

2. *Body preserves invariant:* If the invariant is valid, then it will still be valid after the loop body has been executed.

3. *Usage:* If the loop invariant is valid and the loop guard is invalid, then, after the rest of the surrounding method has been executed, the method's postcondition is valid.

As loops with an unbounded number of iterations cannot be eliminated by symbolic execution (i.e., the application of rules on program fragments) alone and KeY is not able to generate these invariants itself, the user is required to specify every loop in their program.

Again, all three branches depend on the context in which the loop occurs.

While it would be possible to separate the *Body preserves invariant* branch from the context, we will instead implement an alternative to loop invariants.

[Tue12] introduces a rule for loop specification that requires a pre- and postcondition instead of a loop invariant. This rule is based on the observation that a loop can be transformed into a tail-recursive procedure and that finding a pre- and postcondition for this procedure is often easier than finding a loop invariant.

In chapter 5, we will apply this idea in JavaDL, introducing a kind of specialized block contract for blocks that begin with a loop, as well as a rule for the application of loop contracts that, like our block contract rule, allows the user to prove the block's validity separately.

2. Fundamentals

In this chapter, we will give a short introduction to JavaDL and JML.

2.1. Dynamic Logic for Java (*JavaDL*)

2.1.1. Signatures and Kripke Structures

JavaDL is an instance of dynamic logic which integrates Java programs into *JFOL* (*Java First-Order Logic*) formulas [BKW16, 1].

JFOL is an extension of basic first-order logic for Java. It includes a type hierarchy, as well as axioms for integers, heaps, and heap locations [Sch16].

JavaDL formulas are evaluated in *Kripke stuctures.* A Kripke structure is a collection of infinitely many JFOL structures, which we refer to as program states.

A program state $s = (D, \delta, I)$ consists of an interpretation I and a domain (D, δ), consisting of a set D with a typing function δ.

Definition 2.1. [Sch16, 2.1] [BKW16, 2.2] A *JavaDL signature* with respect to a JavaDL type hierarchy $\mathcal{T}$ for a program Prg is a tuple

$$\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym}, \mathrm{ProgVSym})$$

where

1. (FSym, PSym, VSym) is a JFOL signature (consisting of a set of function symbols FSym, a set of predicate symbols PSym, and a set of variable symbols VSym).
2. ProgVSym is the set which contains all local variables declared in Prg as well as some special variables like `heap` (which represents the program's heap) and `self` (which corresponds to the Java variable `this`). For more details on JavaDL's heap semantics, see section 2.1.5.

Definition 2.2. [BKW16, 3.1] A *Kripke structure* for a JavaDL signature Σ is a pair $K = (S, \rho)$ where

1. S is an infinite set of program states such that any two states $s_1, s_2 \in S$ have the same domain (D, δ), and their interpretations only differ for symbols in ProgVSym.

2. ρ is a function such that for every legal program fragment (see section 2.1.2) p and any two states s_1, s_2, $(s_1, s_2) \in \rho(p)$ if any only if p, when started in s_1, terminates in s_2.

Definition 2.3. [BKW16, 2.1, 2.4]

1. DLFml is the set of all JavaDL formulas over a given signature Σ.

2. DLTrm_A is the set of all JavaDL terms of type $A \in \mathcal{T}$ over a given signature Σ for a type hierarchy $\mathcal{T}$.

Definition 2.4. For every JavaDL formula $\phi \in \mathrm{DLFml}$,

1. $\mathrm{var}(\phi) \subseteq \mathrm{VSym}$ is the set of all variables in ϕ.

2. $\mathrm{pvar}(\phi) \subseteq \mathrm{ProgVSym}$ is the set of all program variables in ϕ.

Definition 2.5. Let $\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym}, \mathrm{ProgVSym})$ be a JavaDL signature, $K = (S, \rho)$ a Kripke structure for Σ, $s \in S$ a state, $\beta : \mathrm{VSym} \rightarrow D$ a variable assignment.
Then $\mathrm{val}_{(K,s,\beta)}$ is the *evaluation function* as defined in [ABB+16, 3.3].

2.1.2. Program Fragments

Definition 2.6. [BKW16, 2.3] A *legal program fragment* p in the context of a JavaDL signature Σ is a sequence of Java statements such that there exist local variables $v_1, \ldots, v_n \in \mathrm{ProgVSym}$ with types $T_1, \ldots, T_n$ such that

```
class C {
  static void m(T1 v1,...,Tn vn) throws Throwable { p }
}
```

is a legal program according to the *Java Language Specification* with some extensions. The only one of these extensions used in this thesis is the concept of *method frames*.

Definition 2.7. [BKW16, 2.3] A *method frame* is a statement of the form

```
method-frame(
    result = r,
    source = m(T1,...,Tn)@T,
    this = t ) : {
  body
}
```

where $r \in$ ProgVSym is a local variable, m is a method in the class T, t is an expression free of side effects or method calls, and *body* is a legal program fragment.

Inside the method frame, the visibility rules for the method m apply, and a `return` statement assigns the return value to r and then exits the method frame.

2.1.3. Modalities

For the purpose of including program fragments in a formula, JavaDL contains two modalities, the *diamond modality* $\langle\rangle$, and the *box modality* $[]$, which are defined as follows:

Definition 2.8. [BKW16, 3.2] Let $\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym}, \mathrm{ProgVSym})$ be a JavaDL signature, $K = (S, \rho)$ a Kripke structure for Σ, $s \in S$ a state, $\beta : \mathrm{VSym} \to D$ a variable assignment.

If p is a legal program fragment, and ϕ is a JavaDL formula, then

1. $(K, s, \beta) \models \langle p \rangle \phi$ if and only if $\exists s' \in S : (s, s') \in \rho(p) \land (K, s', \beta) \models \phi$
i.e., if and only if p terminates in a state in which ϕ is true.

2. $(K, s, \beta) \models [p]\phi$ if and only if $\exists s' \in S : (s, s') \in \rho(p) \land (K, s', \beta) \models \phi$ or $\nexists s' \in S : (s, s') \in \rho(p)$
i.e., if and only if p either terminates in a state in which ϕ is true or does not terminate at all.

2.1.4. Type Hierarchy

Before we illustrate JavaDL's heap semantics, we must give a brief overview over JFOL's (and, by extension, JavaDL's) type hierarchy.

A JFOL type hierarchy includes every class type defined in the given Java program. It also includes, among others, the following additional types [Sch16]:

1. $Heap$, the type of the variable `heap` $\in$ ProgVSym

2. $Field$, the type of field references. This is further explained in 2.1.5.

3. *Any*, the parent type of the following types:

   a) *boolean* $\sqsubseteq$ *Any*, which corresponds to Java's `boolean` type. To distinguish between boolean program variables and truth values, we will write the former as TRUE, FALSE and the latter as `true`, `false`.

   b) *int* $\sqsubseteq$ *Any*, which subsumes all integer types in Java.

   c) *Object* $\sqsubseteq$ *Any*, which corresponds to Java's `Object` class.

   d) *LocSet* $\sqsubseteq$ *Any*, a set of heap locations.

2.1.5. Heaps

A variable of type *Heap* maps a value to every heap location (o, f) where o is an *Object* and f is a *Field*. Every JFOL signature contains the following function symbols to operate on heaps [Sch16, 4.1, 4.3]:

1. $\mathrm{select}_A : Heap \times Object \times Field \rightarrow A$ for all $A \sqsubseteq Any$
   This function returns the value of the specified field and is used to translate *Java* statements with a field access `o.f` on the right-hand side.

2. $\mathrm{store} : Heap \times Object \times Field \times Any \rightarrow Heap$
   This function sets the value of the specified field and is used to translate *Java* statements with a field access `o.f` on the left-hand side.

3. $\mathrm{create} : Heap \times Object \rightarrow Heap$
   This function creates a new object on the heap. It is further explained below.

4. $\mathrm{anon} : Heap \times LocSet \times Heap \rightarrow Heap$
   This function anonymizes all heap locations in the specified set. It is further explained in section 2.1.6.

In 2.1.1, we stated that any two program states in the same Kripke structure have the same domain. This is called the *constant domain assumption.* However, in a real Java program, objects may be created at runtime. To get around this, we assume that the domain already contains every object that may be created during the program's runtime with an additional field `created` that is initially set to FALSE. To "create" an object, we simply set its `created` field to TRUE [BKW16, 3.1].

2.1.6. Anonymization

The anon function is defined by the following rule [BKW16, 4.3]:

$$\frac{\begin{array}{l}\text{if } (\epsilon(o, f, s) \wedge f \dot{\neq} \mathtt{created}) \vee \epsilon(o, f, unusedLocs(h)) \\ \text{then } \mathrm{select}_A(h', o, f) \text{ else } \mathrm{select}_A(h, o, f)\end{array}}{\mathrm{select}_A(\mathrm{anon}(h, s, h'), o, f)}$$

where the predicate $\epsilon(o, f, s)$ is `true` if and only if the heap location (o, f) is an element of the *LocSet* s.

If h' is an unknown heap, i.e., a heap which does not occur anywhere else in the sequent, this function returns a heap which assigns unknown values to all locations in the set s (except for `created` fields) and corresponds to h in all other (used) locations.

This function will, for instance, be used for the application of method calls during verification. After a method call, the values of all heap locations in the method's frame (the set of locations whose value the method is allowed to change) are unknown and only restricted by the called method's postcondition.

2.1.7. Updates and Symbolic Execution

As stated in the introduction, KeY applies rules from the sequent calculus to a JavaDL formula to prove the formula's validity. The goal is to simplify the sequent until we are left with one that is trivially true.

Definition 2.9. [Sch16, 2.2] A *sequent* is a pair of sets Γ, Δ of formulas usually denoted in the form

$$\Gamma \implies \Delta$$

Here, Γ is called the *antecedent*, and Δ the *succedent*. The value of such a sequent is equal to the value of the formula

$$\bigwedge_{\gamma \in \Gamma} \gamma \to \bigvee_{\delta \in \Delta} \delta$$

The simplification of program fragments in particular is called *symbolic execution*. The rules for symbolic execution always operate on the *active statement* in a modality.

Definition 2.10. [BKW16, 5.5] The *active statement* in a modality is the first statement in that modality. More specifically, it is the statement after the *non-active prefix* π, which consists of an arbitrary sequence of opening braces `{`, and beginnings of try blocks `try{` and method frames `method-frame(...){`. The rest of the program fragment after the active statement is called the *postfix*, which we denote by ω.

The simplest example for a symbolic execution rule is the following basic assignment rule for assignments without side effects [BKW16, 6.1]:

$$\frac{\Gamma \implies \{loc := value\}\langle \pi\omega \rangle \phi, \Delta}{\Gamma \implies \langle \pi \; \mathtt{loc} = \mathtt{value};\; \omega \rangle \phi, \Delta}$$

Here, $\{loc := value\}$ is an elementary update which assigns the value of the term *value* to the program variable *loc*.

Definition 2.11. [BKW16, 4.1] Let *Prg* be a Java program with a JavaDL type hierarchy $\mathcal{T}$ and a JavaDL signature Σ.

Then the set Upd of updates is inductively defined by:

1. $\{a := t\} \in \mathrm{Upd}$ for every $a : A \in \mathrm{ProgVSym},\ t \in \mathrm{DLTrm}_B$ with $B \sqsubseteq A$
2. $\mathtt{skip} \in \mathrm{Upd}$
   This is the empty update.
3. $u_1, u_2 \in \mathrm{Upd} \implies \{u_1 || u_2\} \in \mathrm{Upd}$
   This parallel update executes u_1, u_2 in parallel.

4. $u_1, u_2 \in \mathrm{Upd} \implies \{\{u_1\}u_2\} \in \mathrm{Upd}$
   This sequential update executes u_2 after u_1.

Alongside modalities with program fragments, updates are another way to denote state changes in JavaDL. The difference between the two is that updates are much more restricted: they only consist of assignments without side effects, and always terminate [BKW16, 4.1]. The goal of symbolic execution is the simplification of a program fragment to an update.

2.2. Java Modeling Language (*JML*)

2.2.1. Method Contracts

JML method contracts are a way to specify a method's behavior.

Let us begin this section with a simple example for a method contract:

```
/*@ normal_behavior
  @ requires arr != null;
  @ ensures (\forall int i; 0 <= i && i < arr.length;
  @     arr[i] == \old(arr[i]) + 1);
  @ assignable arr[*];
  @*/
public static void mapIncrement(int[] arr) {
    int i = 0;
    while (i < arr.length) {
        ++arr[i];
        ++i;
    }
}
```

The keyword `normal_behavior` is short for `signals (Exception) false;` [LPC+13, 9.7], i.e., it guarantees that the method will not throw an exception if the contract's precondition is valid. There is also an `exceptional_behavior` keyword [LPC+13, 9.8], which states that the method always throws an exception if the contract's precondition is valid.

The precondition is described by the `requires` predicate, and the postcondition for normal termination by the `ensures` predicate.

One can also specify a `diverges` predicate to specify under which conditions the method may not terminate.

The frame, i.e., the set of heap locations whose value the method may change, is described by the `assignable` keyword.

The syntax `\old(t)` refers to the value of the term `t` in the method's prestate. Local variables are not affected by `\old`, i.e., for any local variable `v`, `\old(v) == v` is true.

Thus, the above contract states that if the parameter `arr` is not null, `mapIncrement` increments every element in the array by 1, does not throw any exceptions, and does not modify the value of any heap location except for the elements of `arr`.

Here is another example for a method contract:

```
/*@ exceptional_behavior
  @ requires true;
  @ signals (NullPointerException e) true;
  @ signals_only NullPointerException;
  @ assignable \nothing;
  @*/
public static void exceptionalMethod() {
    throw new NullPointerException();
}
```

Here, `true` is the postcondition if the method throws a `NullPointerException`. The `signals_only` clause states that no other type of exception is thrown.

2.2.2. Block Contracts

As will be discussed in chapter 3, it is sometimes helpful to wrap a part of a method in a block with its own pre- and postconditions.

Block contracts, introduced by [Wac12], are a way to achieve this, allowing the user to specify a contract for any code block inside of a method.

Let us begin with the following example, which was adapted from [Wac12]:

```
/*@ requires numbers != null;
  @ ensures 0 <= from && from < numbers.length
  @   && (\before(from) < 0 ==> from == 0)
  @   && (\before(from) >= 0 ==> from == \before(from));
  @ returns \result == 0
  @   && (\before(from) >= numbers.length
  @         || numbers.length == 0);
  @ signals_only \nothing;
  @ assignable \nothing;
  @*/
{
    if (from < 0) {
        from = 0;
    }

    if (from >= numbers.length) {
        return 0;
    }
```

```
}
```

Note that instead of `\old`, we use `\before` to refer to the block's prestate. `\old` can be used in block contracts as well, but it refers to the surrounding method's prestate. This is in contrast to [Wac12] and [ABB+16], where `\old` refers to the block's prestate and `\before` does not exist.

Just as in method contracts, `requires` describes the precondition.

Since a block has more possible types of termination than a method, we need more postconditions.

The keyword `ensures` describes the postcondition if the block terminates normally (a `break` statement with a label that belongs to the block also counts as normal termination). So the contract in our example states that if the precondition is true and the block terminates normally, then `from` is set to 0 if it was less than 0 before and is not changed otherwise.

The semantics of `signals` and `signals_only` is equivalent to its semantics in method contracts. Thus, the block in our example must not throw an exception if its precondition is valid.

The keyword `returns` describes the postcondition if the block terminates because of a `return` statement. Thus, if the block our examples terminates because of the `return` statement, the returned result must be 0 and the value of `numbers.length` before the block must have been either 0 or less than that of `from`.

The keywords `breaks` and `continues` describe the postcondition if the block terminates because of a `break` or `continue` statement without a label.

The keywords `breaks( label )` and `continues( label )` describe the postcondition if the block terminates because of a `break` or `continue` statement with the respective label.

There are also corresponding behavior keywords, namely `return_behavior`, `break_behavior`, and `continue_behavior`.

3. Divide-and-Conquer Strategies

In this chapter, we will discuss existing approaches to the division of a method into smaller parts that are easier to verify.

For this, we introduce the following notation:

Definition 3.1. $(\!|p|\!)$ denotes the translation of a JML expression p to JavaDL.

3.1. Method Contracts

The most frequently used way to divide a method into easier-to-verify parts is to divide it into smaller methods. When a method (which we will refer as the *caller*) calls another method (the *callee*), we can apply the callee's contract to make the caller's proof easier.

Consider the following method:

```
public static void caller() {
    ...
    callee();
    ...
}
```

When symbolic execution reaches the call to `callee`, the proof is split into three branches: One branch proves that `callee`'s precondition is true; the other two prove that, if `callee`'s postcondition is true and the rest of `caller` is executed, `caller`'s postcondition will be true. The validity of `callee`'s contract is proved separately.

The simplest version of this rule (which applies to the box modality and thus does not prove that the caller terminates) is as follows [BKW16, 7.1]:

Definition 3.2. `methodContractPartial`:

$$\frac{\begin{array}{c}\Gamma \implies \{\text{context}\}\{\texttt{setParameters}\}\text{pre}, \Delta \\[1ex] \Gamma, \{\text{context}\}\{\text{remember}\}\{\text{anonymize}\}(\texttt{exception} \doteq \texttt{null}) \\ \implies \{\text{context}\}\{\text{remember}\}\{\text{anonymize}\}(\text{post} \to \{\texttt{lhs} = \texttt{res}\}[\pi\omega]\phi), \Delta \\[1ex] \Gamma, \{\text{context}\}\{\text{remember}\}\{\text{anonymize}\}(\neg\texttt{exception} \doteq \texttt{null}) \\ \implies \{\text{context}\}\{\text{anonymize}\}(\text{post} \to \{\texttt{lhs} = \texttt{res}\}[\pi \texttt{throw exception;} \omega]\phi), \Delta\end{array}}{\begin{array}{c}\Gamma \implies \{\text{context}\} \\ [\pi; \texttt{lhs = target.callee(args);}\, \omega]\phi, \Delta\end{array}}$$

Here, Γ and Δ are arbitrary sets of formulas, pre and post are the callee's pre- and postcondition, ϕ is the caller's postcondition, and $\{\text{context}\}$ is the update created by the symbolic execution.

Some more definitions are necessary to understand this rule:

Definition 3.3. Let `assignableLocations` be the callee's assignable expression, and $\texttt{heap}^{\text{anon}}$ a heap that does not appear anywhere else in the sequent. Then we define

$$\{\text{anonymize}\} := \{\texttt{heap} := \text{anon}(\texttt{heap}, (\!|\texttt{assignableLocations}|\!), \texttt{heap}^{\text{anon}})\}$$

This is a so-called anonymization update. The anonymization heap $\texttt{heap}^{\text{anon}}$ does not occur anywhere else in the formula. In other words, this update assigns unknown values to all locations in the set $(\!|\texttt{assignableLocations}|\!)$. The purpose of this is to ensure that the values of all heap locations that the callee has changed are only restricted by the callee's postcondition.

Definition 3.4. Let $\{p_1, \ldots, p_\lambda\}$ be the set of the callee's parameters. Then we define

$$\{\text{remember}\} := \{\texttt{heap}^{\text{pre}} := \texttt{heap} || p_1^{\text{pre}} := p_1 || \ldots || p_n^{\text{pre}} := p_\lambda\}$$

This remembrance update remembers the heap and the values of all parameters in the callee's prestate. This is necessary to be able to translate \old expressions from the callee's postcondition.

Note that the proof for the callee's contract is not included in `methodContractPartial`. As such, `methodContractPartial` is only sound if the callee's contract is valid and the proof for the caller will only be considered closed when the proof for the callee is also closed.

3.2. Method Contracts for Recursive Methods

The rule described in the previous section is valid for any method call, though if we want to prove that the method also terminates, we must use a version of the rule that applies to the diamond modality.

If the method in question is recursive, we also need to add some additional conditions [GBM+16, 1.4].

A contract for a recursive method may contain a measured_by clause. This clause contains a termination witness, i.e., an integer term whose value in the method's prestate is greater than or equal to 0 and which decreases every time the method calls itself (or makes any method call that leads to indirect recursion). For example, we can prove that the recursive algorithm to compute the nth Fibonacci number always terminates with the following contract:

```
/*@ normal_behavior
  @ requires n >= 1;
  @ measured_by n;
  @*/
public static void fibonacci(int n) {
    if (n == 1 || n == 2) {
        return 1;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

`n` is a valid termination witness for this algorithm because

1. Due to the contract's precondition, the value of `n` must always be greater than or equal to 0 in the prestate.
2. We always pass `n-1` or `n-2` to the recursive calls, which means that the value of `n` decreases with every recursive call.

3.3. Loop Invariants

Loop invariants, like block contracts, are a form of auxiliary specification, allowing us to specify a part of a method, in this case a loop [HAGH16, 9.2].

These specifications, as the name implies, do not consist of pre- and postconditions, but of invariants, i.e., formulas that are valid when the loop is first entered, are preserved by the loop body, and can thus be assumed to still be valid after the loop has terminated.

Aside from these invariants, denoted by the JML keyword `loop_invariant`, a loop specification may also contain an `assignable` clause and a `decreases` clause. The semantics of the `decreases` clause is very similar to that of the `measured_by` clause described in section 3.2. It contains a termination witness, whose value is always greater than or equal to 0 and which decreases every time the loop body is executed. A `decreases` clauses is necessary to prove a loop's termination in KeY.

For example, the specification of the following loop states that before every loop iteration and after the loop has terminated, the following is true:

1. The loop index `i` is between `0` and `arr.length`.
2. All elements whose index is less than `i` have been incremented.
3. All elements whose index is greater than or equal `i` have not been changed.

With this knowledge and the additional knowledge that after the loop has terminated, the loop condition `i < arr.length` must be false, we can prove the method's postcondition, i.e., that the method increments every element in `arr`.

```
/*@ normal_behavior
  @ requires arr != null;
  @ ensures (\forall int i; 0 <= i && i < arr.length;
  @       arr[i] == \old(arr[i]) + 1);
  @ assignable arr[*];
  @*/
public static void mapIncrement(int[] arr) {
    int i = 0;

    /*@ loop_invariant (0 <= i && i <= arr.length);
      @ loop_invariant (\forall int j; 0 <= j && j < i;
      @       arr[j] == \old(arr[j]) + 1);
      @ loop_invariant (\forall int j; i <= j && j < arr.length;
      @       arr[j] == \old(arr[j]));
      @ assignable arr[i .. arr.length];
      @ decreases arr.length - i;
      @*/
    while (i < arr.length) {
        ++arr[i];
        ++i;
    }
}
```

When symbolic execution reaches a loop with a loop invariant, the proof is split into the following three branches [BKW16, 7.2]:

$$\frac{\begin{array}{c}(\mathtt{invariant\_initially\_valid})\\(\mathtt{body\_preserves\_invariant})\\(\mathtt{usage})\end{array}}{\Gamma \implies \{\mathrm{context}\}[\pi;\mathtt{loop};\omega]\phi,\Delta}$$

The first branch proves the validity of the loop invariant before the loop is first entered. The second branch proves that the loop body preserves the invariant. The third branch proves that, if the loop invariant and the negated loop condition are valid, the method's postcondition will be valid after the rest of the method has been executed.

3.4. Block Contracts

Sometimes, dividing a long and complicated method into smaller sub-methods is not possible or practical, e.g., when the method has many local variables that would need to be accessible by multiple sub-methods. In such cases, one can still divide the proof into sub-proofs by using block contracts.

Block contracts also allow us to wrap `if` statements in a block to prevent unnecessary branches (or, more accurately, to deal with the branching `if` statement in the block's validity proof instead of having to split the main proof into two branches).

In this section, we will define (a slightly modified version of) the rule for the application of block contracts from [Wac12, 3.3].

3.4.1. Normal form

Before applying a block contract, it is necessary to transform it into the normal form described in [Wac12, 2.4] (The only difference between our syntax for block contracts and the syntax described in [Wac12] is that we allow block contracts to contain a measuredBy clause. Since measuredBy clauses do not need to be modified during this transformation, we can use the exact transformation as described in [Wac12, 2.4]).

By doing this, we will end up with one or two contracts of the following form:

```
requires requiresPredicate;

ensures ensuresPredicate;
returns returnsPredicate;

breaks breaksPredicate;
breaks (breakLabel₁) breaksPredicate₁;
...
breaks (breakLabelξ) breaksPredicateξ;

continues continuesPredicate;
continues (continueLabel₁) continuesPredicate₁;
...
continues (continueLabelπ) continuesPredicateπ;

signals (Exception e) signalsPredicate;
diverges divergesPredicate;

measured_by measuredByTerm;
assignable assignableLocations;
```

Contracts in this form contain a postcondition for every possible type of termination. Furthermore, they do not contain a behavior keyword [Wac12, 2.4].

If the original contract's `diverges` predicate is trivial (i.e., `true` or `false`), its normal form consists of one contract. Otherwise, its normal form consists of two contracts, one with the predicate `true`, and one with the predicate `false` and the negation of the original predicate added to the `requires` predicate.

This is because JavaDL only supports trivial diverges predicates (with its two modalities $\langle\rangle$ and $[]$).

3.4.2. Translation to JavaDL

In this section, we will summarize the translation of a JML block contract to JavaDL as well as define all JavaDL constructs that are necessary to define the block contract rule `blockContract`.

The general translation of JML expressions to JavaDL is described in [GU16, 1.2].

The translation of block contracts is described in [Wac12, 3.3], though our version differs from [Wac12, 3.3] in a few aspects. Those differences are necessary to implement the semantics of \old and \before as described in section 2.2.2, as well as the rule `blockContractExternal`as described in chapter 4.

We first introduce the following updates:

Definition 3.5. Let $\{v_1, \ldots, v_\nu\}$ be the set of all local variables that are changed by the block. Then we define

$$\{\text{remember}\} := \{\mathtt{heap}^{\text{before}} := \mathtt{heap}||v_1^{\text{before}} := v_1|| \ldots ||v_n^{\text{before}} := v_\nu\}$$

This update is used to translate `\before` expressions.

Definition 3.6. Let $\{p_1, \ldots, p_\lambda\}$ be the set of the surrounding method's parameters. Then we define

$$\{\text{rememberOuter}\} = \{\mathtt{heap}^{\text{pre}} = \mathtt{heap}||p_1^{\text{pre}} = p_1|| \ldots ||p_n^{\text{pre}} = p_\lambda\}$$

This update is used to translate `\old` expressions.

Definition 3.7. Let $\{v_1, \ldots, v_\nu\}$ again be the set of all local variables that are changed by the block. Then we define

$$\begin{aligned}\{\text{anonOut}\} = \{&\mathtt{heap} := \mathtt{anon}(\mathtt{heap}, (\!|\mathtt{assignableLocations}|\!), \mathtt{heap}^{\mathtt{anon}}) \\ &||v_1 := v_1^{\text{anon}}|| \ldots ||v_\nu := v_\nu^{\text{anon}}\}\end{aligned}$$

Definition 3.8. The block contract's precondition pre is defined as follows :

$$\text{pre} = (\!|\mathtt{requiresPredicate}|\!) \land \text{mBy} \land \text{selfCond}$$

where

$$\text{mBy} = \begin{cases} \text{measuredBy}((\!|\backslash\texttt{old}(\texttt{measuredByTerm})|\!)), \\ \qquad \text{if the contract has a measuredBy term} \\ \text{measuredByEmpty, otherwise} \end{cases}$$

$$\text{selfCond} = \begin{cases} \texttt{self} \dot{\neq} \texttt{null} \land \texttt{self.created} \doteq \text{TRUE} \\ \qquad \land \text{exactInstance}_C(\texttt{self}),\ \text{if the surrounding method is not static} \\ \texttt{true},\ \text{otherwise} \end{cases}$$

and C is the type defined by the class which contains the block.

mBy and selfCond are not actually necessary for the rule `blockContract` defined in the following section, as they are always a part of the surrounding method's precondition and thus trivially true. They will, however, become necessary for `blockContractExternal`, the rule defined in chapter 4.

Definition 3.9. The postcondition post is defined as follows:

$$\begin{aligned} \text{post} = & \neg \text{abrupt} \rightarrow (\!|\texttt{ensuresPredicate}|\!) \\ & \land \texttt{broke} \doteq \text{TRUE} \rightarrow (\!|\texttt{breaksPredicate}|\!) \\ & \land \bigwedge_{i=1}^{\xi} \texttt{broke}_i \doteq \text{TRUE} \rightarrow (\!|\texttt{breaksPredicate}_i|\!) \\ & \land \texttt{continued} \doteq \text{TRUE} \rightarrow (\!|\texttt{continuesPredicate}|\!) \\ & \land \bigwedge_{i=1}^{\pi} \texttt{continued}_i \doteq \text{TRUE} \rightarrow (\!|\texttt{continuesPredicate}_i|\!) \\ & \land \texttt{returned} \doteq \text{TRUE} \rightarrow (\!|\texttt{returnsPredicate}|\!) \\ & \land \texttt{signalsPredicate} \end{aligned}$$

where

$$\begin{aligned} \text{abrupt} = & \texttt{broke} \doteq \text{TRUE} \lor \bigvee_{i=1}^{\xi} \texttt{broke}_i \doteq \text{TRUE} \\ & \lor \texttt{continued} \doteq \text{TRUE} \lor \bigvee_{i=1}^{\pi} \texttt{continued}_i \doteq \text{TRUE} \\ & \lor \texttt{returned} \doteq \text{TRUE} \lor \texttt{exception} \dot{\neq} \texttt{null} \end{aligned}$$

To be able to use this postcondition, we must ensure that the program variables we test (broke, continued, etc.) are actually set correctly before we evaluate the postcondition. Therefore, we transform our original block `block` into a modified form `block'`:

Definition 3.10. `block`′ is the following program fragment:

```
method-frame(this=self) : {
    boolean broke = false;
    boolean broke₁ = false;
    ...
    boolean broke_ξ = false;

    boolean continued = false;
    boolean continued₁ = false;
    ...
    boolean continued_ξ = false;

    boolean returned = false;

    Throwable exception = null;

    breakOut: try {
        block^almostSafe
    } catch (Throwable e) {
        exception = e;
    }
}
```

where `block`^{almostSafe} is a block obtained by replacing all `break`, `continue`, and `return` statements in the block (except for those that jump to a location inside the block). For example,

```
break label_i ;
```

is replaced by

```
broke_i = true; break breakOut;
```

Thus, whenever `block` would have terminated abruptly, `block`′ instead sets the respective and then terminates normally.

This transformation allows us to properly evaluate the block's postcondition. However, our transformed program is now no longer equivalent to the original program because we have eliminated every kind of abrupt termination.

To solve this problem, we define another program fragment that checks all termination flags and then terminates in the same way `block` would have terminated:

Definition 3.11. `ifCascade` is the following program fragment:

```
if (broke) break;
if (broke₁) break label₁;
...
if (broke_ξ) break label_ξ;

if (continued) continue;
if (continued₁) continue label₁;
...
if (continued_π) continue label_π;

if (returned) return result;
    // or, if the surrounding method is void:
    // if (returned) return;
if (exception != null) throw exception;
```

For the remaining sub-terms and sub-formulas, we do not give exact definitions, instead referring to [Wac12, 3.3].

reachableIn ensures that all free reference variables in the block are reachable, i.e., that they are either `null` or a reference to a valid heap location.

reachableOut ensures the same for all reference variables that may be changed by `block`′.

frame ensures that no heap locations that are not in the set `assignableLocations` have been changed.

atMostOneFlagSet ensures that at most one abrupt termination flag is set (see abrupt from Definition 3.9).

3.4.3. Rule

The complete rule for the application of block contracts is as follows:

Definition 3.12. `blockContract`:

$$\frac{\begin{array}{c}\Gamma \implies \{\text{rememberOuter}\}\{\text{context}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn} \\ \rightarrow \{\text{remember}\}[\![\texttt{block}']\!](\text{post} \wedge \text{frame})\ (1) \\ \Gamma \implies \{\text{rememberOuter}\}\{\text{context}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn}), \Delta\ (2) \\ \Gamma \implies \{\text{rememberOuter}\}\{\text{context}\}\{\text{remember}\}\{\text{anonOut}\} \\ (\text{post} \wedge \text{wellFormed}(\texttt{heap}^{\text{anon}}) \wedge \text{reachableOut} \wedge \text{atMostOneFlagSet} \\ \rightarrow [\![\pi; \texttt{ifCascade}; \omega]\!]'\phi), \Delta\ (3)\end{array}}{\Gamma \implies \{\text{rememberOuter}\}\{\text{context}\}[\![\pi; \texttt{block}; \omega]\!]'\phi, \Delta}$$

where

$$[\![]\!]' \begin{cases} \in & \{\langle\rangle, []\} & , \text{ if } [\![]\!] = \langle\rangle \\ = & [] & , \text{ if } [\![]\!] = [] \end{cases}$$

Definition 3.13. From now on, we will refer to the premisses of the above rule by the following names:

1. (`valid`)
2. (`pre`)
3. (`usage`)

The first premiss (`valid`) ensures that the block contract is valid in the context of the surrounding method.

The second premiss (`pre`) ensures that the block's precondition is true when the block is executed.

The third premiss (`usage`) ensures that, if the block's postcondition is true, then the surrounding method's postcondition will be true after the rest of the method has been executed.

4. A New Block Contract Rule

Our goal in this chapter is the introduction of an alternative rule `blockContractExternal` for block contracts, which allows us to prove the validity of a block contract in a separate proof obligation.

The other two premisses, (`pre`) and (`usage`), will remain unchanged from `blockContract`.

4.1. Replacing the Context

Let us recall the first premiss (`valid`) from `blockContract`:

$$\Gamma \implies \{\text{rememberOuter}\}\{\text{context}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn} \\ \rightarrow \{\text{remember}\}[\![\texttt{block}']\!](\text{post} \wedge \text{frame})$$

The surrounding method's state when the block is entered is encoded in two parts of this premiss: firstly, the update {context}, which is generated from the symbolic execution of the code before the block, and secondly, the antecedent Γ, which contains the surrounding method's precondition as well as several other conditions that arose during symbolic execution. It is thus those two parts we need to replace.

The update {context} is replaced by the following anonymizing update:

Definition 4.1. Let $\{v_1, \ldots, v_n\}$ be the set of all program variables that occur in the block or its contract, and `allLocs` the set of all heap locations. Then we define

$$\{\text{anonIn}\} := \{\texttt{heap} := \text{anon}(\texttt{heap}, \texttt{allLocs}, \texttt{heap}^{\text{anon}}) || v_1 := v_1^{\text{anon}} || \ldots || v_\mu := v_\mu^{\text{anon}}\}$$

This update sets every variable v_i that would have occurred in context to an unknown value v_i^{anon}. To ensure that the variable `self` still refers to a valid object in the state after this update, we must introduce additional preconditions for `self`, which we did in section 3.4.2.

The antecedent Γ is replaced by the predicate

$$\text{wellFormed}(\texttt{heap}), \text{wellFormed}(\texttt{heap}^{\text{anon}})$$

The predicate wellFormed(`heap`) contains restrictions to `heap` that correct some of the over-generalizations in JavaDL's heap model [Sch16, 4.3]. These restrictions are necessary to prove a block contract's validity.

Intuitively, the soundness of `blockContractExternal` immediately follows from the soundness of `blockContract`. The validity premiss in `blockContract`, (`valid`), states that the block contract is valid in the surrounding method's context. By replacing context with anonIn, we instead state that it is valid in *any* context.

4.2. Recursion

An issue with this rule occurs if the surrounding method is recursive, and a recursive call occurs inside of the block.

Consider the following example:

```
/*@ normal_behavior
  @ requires idx <= arr.length && idx >= 0;
  @ ensures \result == arr.length - idx;
  @ measured_by arr.length - idx;
  @*/
public static int lengthFrom(int[] arr, int idx) {
    if (idx == arr.length) {
        return 0;
    } else {
        ++idx;
        /*@ return_behavior
          @ requires arr != null;
          @ requires idx <= arr.length && idx >= 0;
          @ returns \result == arr.length - idx + 1;
          @*/
        {
            return lengthFrom(arr, idx) + 1;
        }
    }
}
```

If we try to prove this block contract's validity with `blockContractExternal`, we have no way of knowing whether the recursive call will terminate. This is because in replacing Γ, we removed the surrounding method's `measured_by` clause.

We cannot simply add this clause back in because the surrounding method may have multiple contracts with different `measured_by` clauses.

To solve this problem, we must add a `measured_by` clause – along with some additional `requires` predicates to restrict the value of the `measured_by` term in the surrounding method's prestate – to the block contract.

This is why we added the `measured_by` predicate to the block's precondition in section 3.4.2: We must ensure that a block contract can only be applied if it has the same `measured_by` clause as the method contract we are trying to prove; otherwise, we would be able to prove the termination of non-terminating methods.

The complete block contract is as follows:

```
/*@ return_behavior
  @ requires arr != null;
  @ requires idx <= arr.length && idx >= 0;
  @ requires \old(arr.length - (idx + 1))
  @     == arr.length - idx;
  @ requires \old(arr.length - idx) > 0;
  @ returns \result == arr.length - idx + 1;
  @ measured_by arr.length - idx;
  @*/
{
    return lengthFrom(arr, idx) + 1;
}
```

The additional `requires` predicates we added are necessary to prove termination: As part of the precondition of the recursive call we must show that the value of its `measured_by` term is nonnegative and less than the value in the surrounding method's prestate [GBM+16, 1.4].

4.3. Rule

The changes described above yield the following new rule:

Definition 4.2. `blockContractExternal`:

$$\dfrac{\begin{array}{l} \mathrm{wellFormed}(\mathtt{heap}), \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \\ \quad \Longrightarrow \{\mathrm{rememberOuter}\}\{\mathrm{anonIn}\}(\mathrm{pre} \land \mathrm{wellFormed}(\mathtt{heap}) \land \mathrm{reachableIn} \\ \qquad \to \{\mathrm{remember}\}[\![\mathtt{block}']\!](\mathrm{post} \land \mathrm{frame})\ (1) \\ \Gamma \Longrightarrow \{\mathrm{rememberOuter}\}\{\mathrm{context}\}(\mathrm{pre} \land \mathrm{wellFormed}(\mathtt{heap}) \land \mathrm{reachableIn}), \Delta\ (2) \\ \quad \Gamma \Longrightarrow \{\mathrm{rememberOuter}\}\{\mathrm{context}\}\{\mathrm{remember}\}\{\mathrm{anonOut}\} \\ \qquad (\mathrm{post} \land \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \land \mathrm{reachableOut} \land \mathrm{atMostOneFlagSet} \\ \qquad\quad \to [\![\pi; \mathtt{ifCascade}; \omega]\!]'\phi), \Delta\ (3) \end{array}}{\Gamma \Longrightarrow \{\mathrm{rememberOuter}\}\{\mathrm{context}\}[\![\pi; \mathtt{block}; \omega]\!]'\phi, \Delta}$$

Definition 4.3. The first premiss of the above rule is called (`valid`*).

The second and third premiss, (`pre`) and (`usage`), are the same as those in `blockContract`.

4.4. Proof of Soundness

Since `blockContract` is sound [Wac12, 3.4] and we have only modified the premiss (`valid`), it suffices to reduce our modified premiss (`valid`*) to (`valid`).

We first establish the following theorem, which essentially states that the validity of a sequent $\Psi \Longrightarrow \{\mathrm{context}\}\phi$ follows from the validity of $\Psi \Longrightarrow \{\mathrm{anonymize}\}\phi$, with $\Psi, \phi\{\mathrm{context}\}, \{\mathrm{anonymize}\}$ as defined below.

Theorem (Context Replacement):

Let $\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym}, \mathrm{ProgVSym})$ be a JavaDL signature.

Let Ψ be a set of JavaDL ground formulas for Σ, ϕ a JavaDL formula for Σ, anonymize := $\{v_1 := v_1^{\mathrm{anon}} || \ldots || v_n := v_n^{\mathrm{anon}}\}$ where

$$\{v_1, \ldots, v_n\} \subseteq \mathrm{pvar}(\phi), \tag{4.1}$$

$$\{v_1^{\mathrm{anon}}, \ldots, v_n^{\mathrm{anon}}\} \subseteq \mathrm{FSym} \tag{4.2}$$

$$v_1^{\mathrm{anon}}, \ldots, v_n^{\mathrm{anon}} \text{ appear neither in } \Psi \text{ nor in } \phi \tag{4.3}$$

such that $\models \Psi \Longrightarrow \{\mathrm{anonymize}\}\phi$ holds.

Let context := $\{v_1 := t_1 || \ldots || v_m := t_m\}$ where

$$\{t_1, \ldots, t_m\} \subseteq \mathrm{DLTrm}_{Any} \tag{4.4}$$
$$m \geq n \tag{4.5}$$
$$\{v_1, \ldots, v_m\} \cap \mathrm{pvar}(\phi) = \{v_1, \ldots, v_n\} \tag{4.6}$$

Then $\models \Psi \implies \{\text{context}\}\phi$ holds.

Proof:

Let $K_c = (S_c, \rho_c)$ be a Kripke structure for Σ, $s_c^1 \in S_c$ a state, $\beta : \mathrm{VSym} \to D$ a variable assignment, and $s_c^2 := \mathrm{val}_{(K_c, s_c^1, \beta)}(\text{context})(s_c^1)$.

Let $K_a = (S_a, \rho_a)$ be another Kripke structure for Σ, $s_a^1 \in S_a$, $s_a^2 := \mathrm{val}_{(K_a, s_a^1, \beta)}(\text{anonymize})(s_a^1)$, such that

$$\forall i \in \{1, \ldots, n\} : \ \mathrm{val}_{(K_a, s_a^1, \beta)}(v_i^{\text{anon}}) = \mathrm{val}_{(K_c, s_c^1, \beta)}(t_i) \tag{4.7}$$

$$\forall \sigma \in \mathrm{FSym} \cup \mathrm{PSym} \cup \mathrm{VSym} \cup \mathrm{ProgVSym} \setminus \{v_1^{\text{anon}}, \ldots, v_n^{\text{anon}}\} : \ \mathrm{val}_{(K_a, s_a^1, \beta)}(\sigma) = \mathrm{val}_{(K_c, s_c^1, \beta)}(\sigma) \tag{4.8}$$

Since $\models \Psi \implies \{\text{anonymize}\}\phi$ holds, $(K_a, s_a^1, \beta) \models \Psi \implies \{\text{anonymize}\}\phi$ holds too.

If $(K_a, s_a^1, \beta) \models \neg\Psi$ holds, then – due to 4.3 and 4.8 – $(K_c, s_c^1, \beta) \models \neg\Psi$ holds too.

We know that $(K_a, s_a^1, \beta) \models \{\text{anonymize}\}\phi$ holds if and only if $(K_a, s_a^2, \beta) \models \phi$ holds.

From 4.7, we get

$$\forall i \in \{1, \ldots, n\} : \ \mathrm{val}_{(K_a, s_a^2, \beta)}(v_i) = \mathrm{val}_{(K_c, s_c^2, \beta)}(v_i) \tag{4.9}$$

Due to 4.3 and 4.8, we know that $\mathrm{val}_{(K_a, s_a^1, \beta)}(\sigma) = \mathrm{val}_{(K_c, s_c^1, \beta)}(\sigma)$ for every σ that appears in ϕ. Due to 4.6 and 4.9, we also know that $\mathrm{val}_{(K_a, s_a^2, \beta)}(\sigma) = \mathrm{val}_{(K_c, s_c^2, \beta)}(\sigma)$ for every σ that appears in ϕ.

Thus, $\mathrm{val}_{(K, s_c^2, \beta)}(\phi) = \mathrm{val}_{(K, s_a^2, \beta)}(\phi)$, i.e., $(K, s_c^2, \beta) \models \phi$ holds if and only if $(K, s_a^2, \beta) \models \phi$ holds. Therefore $(K, s_c^1, \beta) \models \{\text{context}\}\phi$ holds if and only if $(K, s_a^1, \beta) \models \{\text{anonymize}\}\phi$ holds.

Altogether, we have proven that $(K_c, s_c^1, \beta) \models \Psi \implies \{\text{context}\}\phi$ holds. Because (K_c, s_c^1, β) was chosen arbitrarily, this implies that $\models \Psi \implies \{\text{context}\}\phi$ holds.

■

This theorem assumes that all anonymization functions v_i^{anon} are nullary, i.e., constants. However, anonIn also uses the trinary function $\mathtt{anon}(h_1, l, h_2)$.

Thus, instead of applying the theorem directly, we apply the following corollary:

Corollary:

Let

$$\text{anonIn} = \{\texttt{heap} := \text{anon}(\texttt{heap}, \texttt{allLocs}, \texttt{heap}^{\text{anon}}) || \text{anonymize}\}$$

and anonymize, context, Ψ, ϕ as above, such that $\texttt{heap}^{\text{anon}}$ does not appear in Ψ, context, ϕ.

Then we can deduce

$$\cfrac{\cfrac{\cfrac{\models \Psi, \text{wellFormed}(\texttt{heap}^{\text{anon}}) \implies \{\text{anonIn}\}\phi}{\begin{array}{l}\models \Psi, \text{wellFormed}(\texttt{heap}^{\text{anon}}), \texttt{heap}^{\text{anon}} = \texttt{heap} \\ \qquad \implies \{\text{anonymize}\}\{\texttt{heap} := \text{anon}(\texttt{heap}, \texttt{allLocs}, \texttt{heap}^{\text{anon}})\}\phi\end{array}}}{\begin{array}{l}\models \Psi, \text{wellFormed}(\texttt{heap}^{\text{anon}}), \texttt{heap}^{\text{anon}} = \texttt{heap} \\ \qquad \implies \{\text{context}\}\{\texttt{heap} := \text{anon}(\texttt{heap}, \texttt{allLocs}, \texttt{heap}^{\text{anon}})\}\phi\end{array}}\ \text{Theorem}}{\models \Psi \implies \{\text{context}\}\phi}\ \texttt{heap}^{\text{anon}} = \texttt{heap}$$

Using this corollary, and the fact that no remembrance variables occur in the antecedents $\text{wellFormed}(\texttt{heap}) \wedge \text{wellFormed}(\texttt{heap}^{\text{anon}})$ and Γ, we can now prove that for any Kripke structure $K = (S, \rho)$ and variable assignment β with $s^1 \in S$ and $s^2 := \text{val}_{(K,s,\beta)}(\text{rememberOuter})(s^1)$:

$$\cfrac{\begin{array}{l}(K, s^1, \beta) \models \text{wellFormed}(\texttt{heap}), \text{wellFormed}(\texttt{heap}^{\text{anon}}) \\ \qquad \implies \{\text{rememberOuter}\}\{\text{anonIn}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn} \\ \qquad\quad \rightarrow \{\text{remember}\}[\![\texttt{block}']\!](\text{post} \wedge \text{frame})\end{array}}{\cfrac{\cfrac{\begin{array}{l}(K, s^2, \beta) \models \text{wellFormed}(\texttt{heap}), \text{wellFormed}(\texttt{heap}^{\text{anon}}) \\ \qquad \implies \{\text{anonIn}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn} \\ \qquad\quad \rightarrow \{\text{remember}\}[\![\texttt{block}']\!](\text{post} \wedge \text{frame})\end{array}}{\begin{array}{l}(K, s^2, \beta) \models \text{wellFormed}(\texttt{heap}) \\ \qquad \implies \{\text{context}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn} \\ \qquad\quad \rightarrow \{\text{remember}\}[\![\texttt{block}']\!](\text{post} \wedge \text{frame})\end{array}}\ \text{Corollary}}{\cfrac{}{\begin{array}{l}(K, s^2, \beta) \models \Gamma \\ \qquad \implies \{\text{context}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn} \\ \qquad\quad \rightarrow \{\text{remember}\}[\![\texttt{block}']\!](\text{post} \wedge \text{frame})\end{array}}}\ \text{wellFormed}(\texttt{heap}) \in \Gamma}}$$

$$\begin{array}{l}(K, s^1, \beta) \models \Gamma \\ \qquad \implies \{\text{rememberOuter}\}\{\text{context}\}(\text{pre} \wedge \text{wellFormed}(\texttt{heap}) \wedge \text{reachableIn} \\ \qquad\quad \rightarrow \{\text{remember}\}[\![\texttt{block}']\!](\text{post} \wedge \text{frame})\end{array}$$

■

5. Loop Contracts

Our goal in this chapter is to provide an alternative to loop invariants based on block contracts.

As stated in the introduction, [Tue12] introduces an alternative rule for loops that requires a pre- and postcondition instead of a loop invariant. This rule is based on the observation that a loop can be transformed into a tail-recursive procedure and that in some cases finding a contract for this procedure is easier than finding an invariant for a loop. We will confirm this observation in chapter 7.

For instance, a block:

```
{
    while (loopCondition) {
        body;
    }
    tail;
}
```

is equivalent to the following tail-recursive procedure:

```
procedure(args) {
    if (loopCondition) {
        body;
        procedure(args);
    } else {
        tail;
    }
}
```

[Tue12]'s rule is for separation logic, but we can apply the same idea in JavaDL.

Before we do so however, we will define the JML syntax and semantics for our loop contracts.

5.1. Syntax

A loop contract is a contract for a block with structure

```
{
    while (loopCondition) {
```

```
        body
    }
    tail
}
```

where `body` and `tail` are arbitrary sequences of of Java statements and `loopCondition` is an expression of type `boolean`.

The syntax for a loop contract is the same as for a block contract, except for the following differences:

- Every loop contract must start with the keyword `loop_contract`.
- `decreases` clauses are permitted (and necessary to prove termination!)

5.2. Semantics

The semantics for loop contracts differs in the following ways from block contracts:

In a block contract, the precondition must only hold when the block is first entered. In a loop contract, the precondition must hold every time the loop is repeated.

In a block contract, the `assignable` set must contain all variables that may be changed during the block's execution. In a loop contract, the assignable set may exclude variables that were changed during previous loop iterations. Similarly, `\before(t)` refers to the value of `t` before the current loop iteration.

All of this becomes more clear with an example:

```
/*@ loop_contract normal_behavior
  @ requires arr != null && 0 <= i && i <= arr.length;
  @ ensures (\forall int j; \before(i) <= j && j < arr.length;
  @           arr[j] == \before(arr[j]) + 1);
  @ assignable arr[i .. arr.length];
  @ decreases arr.length - i;
  @*/
{
    while (i < arr.length) {
        ++arr[i];
        ++i;
    }
}
```

This contract states that before every loop iteration `arr != null && 0 <= i && i <= arr.length` holds and that when the loop has terminated, every array element whose index is greater than or equal to `i` will be incremented.

The `assignable` clause states that no heap locations except for those array elements are modified.

As with loop invariants (see section 3.3), the `decreases` clause serves as a termination witness.

5.3. Normal Form

Before we translate JML loop contract to JavaDL, we transform it to its normal form.

This transformation is equivalent to the one described in section 3.4.1. `decreases` clauses are not changed during this transformation.

As in section 3.4.1, we end up with one (if the `diverges` predicate was trivial) or two (otherwise) loop contracts.

5.4. Translation to *JavaDL*

As in section 3.4.2, it is necessary to transform *body* and *tail* into the modified forms $body^{almostSafe}$, $tail^{almostSafe}$ by replacing all `return`s as well as `break`s and `continue`s without labels or with labels that do not occur inside of the block.

In addition, we replace any statements of the form `break label;` where `label` belongs to the loop, as well as regular `break;` statements, inside of the loop body with

`brokeLoop = true; break breakLoop;`

Any statements of the form `continue label;` where `label` belongs to the loop, as well as regular `continue;` statements, inside of the loop body are replaced by

`break breakLoop;`

Definition 5.1. body′ is the following program fragment:

```
method-frame(this=self) : {
    boolean brokeLoop = false;

    boolean broke_1 = false;
    ⋮
    boolean broke_ξ = false;

    boolean continued_1 = false;
    ⋮
    boolean continued_ξ = false;

    boolean returned = false;
```

```
    Throwable exception = null;

    breakOut: breakLoop: try {
        body^almostSafe
    } catch (Throwable e) {
        exception = e;
    }
}
```

Definition 5.2. `tail′` is the following program fragment:

```
method-frame(this=self) : {
    boolean broke = false;
    boolean broke_1 = false;
    ⋮
    boolean broke_ξ = false;

    boolean continued_1 = false;
    ⋮
    boolean continued_ξ = false;

    boolean returned = false;

    Throwable exception = null;

    breakOut: try {
        tail^almostSafe
    } catch (Throwable e) {
        exception = e;
    }
}
```

Definition 5.3. `unfold′` is the following program fragment:

```
method-frame(this=self) : {
    try {
        cond = loopCodition;
    } catch (Throwable e) {
        exception = e;
    }
}
```

where cond : *boolean* $\in$ ProgVSym.

Definition 5.4. A loop contract's precondition pre is defined in the same way as a block contract's precondition in 3.4.2. In addition, if the loop contract has a decreases term, the predicate $(\!|\texttt{decreasesTerm}|\!) \geq 0$ is added to the precondition.

Definition 5.5. If the contract has a decreases term, decreasesCheck is the predicate $(\!|\texttt{decreasesTerm}|\!) < (\!|\texttt{\textbackslash before(decreasesTerm)}|\!)$; otherwise decreasesCheck $= \texttt{true}$.

Definition 5.6. The modality $[\![]\!]$ is defined as follows:

$$[\![]\!] = \begin{cases} \langle\rangle & , \quad \text{if a } \texttt{decreases} \text{ term is specified and the } \texttt{diverges} \text{ predicate is } \texttt{false} \\ [] & , \quad \text{otherwise} \end{cases}$$

5.5. A Simplified Rule

We have already mentioned that loop contracts are based on the observation that a block than starts with a loop can be transformed into a recursive method. This means that we need an equivalent to the recursive method call, i.e., a way of applying the loop contract for the subsequent loop iteration.

In a recursive method, we assume that the recursive call's postcondition is true and prove that this implies our postcondition.

Similarly, in a loop contract, we need to assume that the next iteration's postcondition is true and prove that this implies the current iteration's postcondition.

For this, we define two remembrance updates:

Definition 5.7.

$$\mathrm{remember}_{\mathrm{current}} = \mathrm{remember} = \{\texttt{heap}^{\mathrm{before}} = \texttt{heap} || v_1^{\mathrm{before}} = v_1 || \dots || v_n^{\mathrm{before}} = v_n\}$$
$$\mathrm{remember}_{\mathrm{next}} = \{\texttt{heap}^{\mathrm{beforeN}} = \texttt{heap} || v_1^{\mathrm{beforeN}} = v_1 || \dots || v_n^{\mathrm{beforeN}} = v_n\}$$

We also have two postconditions $\mathrm{post}_{\mathrm{current}}$, $\mathrm{post}_{\mathrm{next}}$. Both are translated from the JML contract like in 3.4.2, but they differ in the way `\before` expressions are translated: In $\mathrm{post}_{\mathrm{current}}$, `\before` expressions refer to the state remembered by $\mathrm{remember}_{\mathrm{current}}$, and in $\mathrm{post}_{\mathrm{next}}$, they refer to the state remembered by $\mathrm{remember}_{\mathrm{next}}$.

The same is true for the two framing conditions $\mathrm{frame}_{\mathrm{current}}$, $\mathrm{frame}_{\mathrm{next}}$.

We also define the following anonymization update:

Definition 5.8. Let $\{v_1, \dots, v_n\}$ be the set of all local variables that are modified in $\texttt{unfold}'$ or in $\texttt{body}'$. Then

$$\begin{aligned} \mathrm{anonOut}_{\mathrm{loop}} := \{&\texttt{heap} := \mathrm{anon}(\texttt{heap}, (\!|\texttt{assignableLocations}|\!), \texttt{heap}^{\mathrm{anon}}) \\ &|| v_1 := v_1^{\mathrm{anon}} || \dots || v_n := v_n^{\mathrm{anon}}\} \end{aligned}$$

With these definitions out of the way, we can specify a simplified version of our loop contract rule. This rule is equivalent to the final rule presented in section 5.6, except for the fact that, for simplicity's sake, it does not handle abrupt termination.

Before we specify the rule, let us think about what it needs to do. The rule must cover the following two cases:

1. If the loop condition is false, the tail must be executed and then the postcondition must hold.
2. If the loop condition is true, the body must be executed. After the body has terminated, the decreases check and the precondition must hold. Then the contract of the next loop iteration must be applied.

Thus, we end up with the following rule:

$$
\frac{
\begin{array}{l}
\mathrm{wellFormed}(\mathtt{heap}), \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \implies \{\mathrm{rememberOuter}\}\{\mathrm{anonIn}\}(\\
\quad \mathrm{pre} \land \mathrm{wellFormed}(\mathtt{heap}) \land \mathrm{reachableIn} \\
\quad\quad \rightarrow \{\mathrm{remember}_{\mathrm{current}}\}[\![\mathtt{unfold}']\!](\\
\quad\quad\quad \mathrm{cond} \doteq \mathrm{FALSE} \rightarrow [\![\mathtt{tail}']\!](\mathrm{post}_{\mathrm{current}} \land \mathrm{frame}_{\mathrm{current}}) \\
\quad\quad\quad \land(\mathrm{cond} \doteq \mathrm{TRUE} \rightarrow [\![\mathrm{body}']\!](\mathrm{decreasesCheck} \land \mathrm{pre} \\
\quad\quad\quad\quad \land \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{pre}}) \land \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \\
\quad\quad\quad\quad \land \{\mathrm{remember}_{\mathrm{next}}\}\{\mathrm{anonOut}_{\mathtt{loop}}\}[\![\mathtt{tail}']\!] \\
\quad\quad\quad\quad\quad (\mathrm{post}_{\mathrm{next}} \land \mathrm{frame}_{\mathrm{next}} \rightarrow \mathrm{post}_{\mathrm{current}} \land \mathrm{frame}_{\mathrm{current}}) \\
\quad\quad\quad) \\
\quad\quad) \\
\quad) \\
\Gamma \implies \{\mathrm{rememberOuter}\}\{\mathrm{context}\}(\mathrm{pre} \land \mathrm{wellFormed}(\mathtt{heap}) \land \mathrm{reachableIn}), \Delta \\
\Gamma \implies \{\mathrm{rememberOuter}\}\{\mathrm{context}\}\{\mathrm{remember}\}\{\mathrm{anonOut}_{\mathtt{all}}\} \\
\quad\quad (\mathrm{post} \land \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \land \mathrm{reachableOut} \land \mathrm{atMostOneFlagSet} \\
\quad \rightarrow [\![\mathtt{nonActivePrefix}; \mathtt{ifCascade}; \mathtt{afterBlock}]\!]'\mathrm{methodPost}), \Delta
\end{array}
}{
\Gamma \implies \{\mathrm{rememberOuter}\}\{\mathrm{context}\}[\![\pi; \mathtt{block}; \omega]\!]'\phi, \Delta
}
$$

The second and third premiss, (pre) and (usage), remain unchanged from blockContract.

5.6. Rule

To make sure our rule can also handle abrupt termination, we add some more case distinctions. The final rule must cover all of the following cases:

1. If an uncaught exception is thrown during the evaluation of the loop condition, the block terminates. Thus, the postcondition must hold immediately after the evaluation.

2. If the loop condition evaluates to false, the loop body is skipped and the tail is executed. Then, the postcondition must hold.

3. If the loop condition evaluates to true, the loop body is executed.
   a) If the loop body terminates due to a `break` statement, and this break statement only jumps out of the body (and not out of the entire block), the tail is executed, and then the postcondition must hold.
   b) If the execution of the loop body leads to an abrupt termination of the block (e.g., if the loop body throws an uncaught exception, or contains a `break` statement that leads to a label outside the block), the postcondition must hold after the body has terminated.
   c) If the loop body terminates normally, the `decreases` check and the precondition must hold. Then, the contract of the next loop iteration must be applied.
      i. If the subsequent loop iterations terminate abruptly, the postcondition must hold after they have terminated.
      ii. Otherwise, the tail is executed, and then the postcondition must hold.

We end up with the following rule:

Definition 5.9. `loopContract`:

$$
\begin{array}{l}
\mathrm{wellFormed}(\mathtt{heap}), \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \implies \{\mathrm{rememberOuter}\}\{\mathrm{anonIn}\}(\\
\quad \mathrm{pre} \land \mathrm{wellFormed}(\mathtt{heap}) \land \mathrm{reachableIn} \\
\qquad \to \{\mathrm{remember}_{\mathrm{current}}\}[\![\mathtt{unfold}']\!](\\
\qquad\quad (\mathtt{exception}\dot{\neq}\mathtt{null} \to (\mathrm{post}_{\mathrm{current}} \land \mathrm{frame}_{\mathrm{current}})) \\
\qquad\quad \land(\mathtt{exception}\dot{=}\mathtt{null} \land \mathtt{cond}\dot{=}\mathtt{FALSE} \to [\![\mathtt{tail}']\!](\mathrm{post}_{\mathrm{current}} \land \mathrm{frame})) \\
\qquad\quad \land(\mathtt{exception}\dot{=}\mathtt{null} \land \mathtt{cond}\dot{=}\mathtt{TRUE} \\
\qquad\qquad \to [\![\mathrm{body}']\!](\\
\qquad\qquad\quad (\mathtt{brokeLoop}\dot{=}\mathtt{TRUE} \to [\![\mathtt{tail}']\!](\mathrm{post}_{\mathrm{current}} \land \mathrm{frame}_{\mathrm{current}})) \\
\qquad\qquad\quad \land(\mathrm{abrupt} \to \mathrm{post}_{\mathrm{current}} \land \mathrm{frame}_{\mathrm{current}}) \\
\qquad\qquad\quad \land(\mathtt{brokeLoop}\dot{=}\mathtt{FALSE} \land \neg\mathrm{abrupt} \to \mathrm{pre} \land \mathrm{decreasesCheck} \\
\qquad\qquad\qquad \land\mathrm{wellFormed}(\mathtt{heap}^{\mathrm{pre}}) \land \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \\
\qquad\qquad\qquad \land\{\mathrm{remember}_{\mathrm{next}}\}\{\mathrm{anonOut}_{\mathtt{loop}}\}(\\
\qquad\qquad\qquad\quad (\mathrm{abrupt} \to (\mathrm{post}_{\mathrm{next}} \land \mathrm{frame}_{\mathrm{next}} \to \mathrm{post}_{\mathrm{current}} \land \mathrm{frame}_{\mathrm{current}}) \\
\qquad\qquad\qquad\quad \land(\neg\mathrm{abrupt} \to [\![\mathtt{tail}']\!] \\
\qquad\qquad\qquad\qquad (\mathrm{post}_{\mathrm{next}} \land \mathrm{frame}_{\mathrm{next}} \to \mathrm{post}_{\mathrm{current}} \land \mathrm{frame}_{\mathrm{current}}) \\
\qquad\qquad\qquad\quad) \\
\qquad\qquad\qquad) \\
\qquad\qquad\quad) \\
\qquad\qquad) \\
\qquad\quad) \\
\qquad) \\
\quad) \ (1) \\
\quad \Gamma \implies \{\mathrm{rememberOuter}\}\{\mathrm{context}\}(\mathrm{pre} \land \mathrm{wellFormed}(\mathtt{heap}) \land \mathrm{reachableIn}), \Delta \ (2) \\
\quad\ \Gamma \implies \{\mathrm{rememberOuter}\}\{\mathrm{context}\}\{\mathrm{remember}\}\{\mathrm{anonOut}_{\mathtt{all}}\} \\
\qquad\qquad (\mathrm{post} \land \mathrm{wellFormed}(\mathtt{heap}^{\mathrm{anon}}) \land \mathrm{reachableOut} \land \mathrm{atMostOneFlagSet} \\
\qquad \to [\![\mathtt{nonActivePrefix}; \mathtt{ifCascade}; \mathtt{afterBlock}]\!]'\mathrm{methodPost}), \Delta \ (3) \\
\hline
\qquad\qquad \Gamma \implies \{\mathrm{rememberOuter}\}\{\mathrm{context}\} \\
\qquad\qquad\quad [\![\pi; \mathtt{block}; \omega]\!]'\phi, \Delta
\end{array}
$$

Definition 5.10. The first premiss of the above rule is called (`valid_loop`).

The second and third premiss, (`pre`) and (`usage`), are the same as those in `blockContract`.

5.7. Soundness Proof Sketch

Theorem (Soundness):

`loopContract` is sound.

Proof sketch:

Instead of the full proof, we present a short proof sketch. The full proof can be found in appendix A.

Recall the case distinction at the beginning of the section 5.6.

One can quickly see that (`valid_loop`) covers all of the simple cases, i.e., all cases that do not involve the application of the contract for the next iteration.

It remains to be shown that the loop contract for the next iteration is applied correctly. For this we observe the following: If (`valid_loop`) is universally valid, then after every loop iteration either the block has terminated or the decreases check and the preconditions are preserved. Furthermore, every loop iteration's postcondition implies the previous iteration's postcondition. Thus, we can show by induction that the first iteration's postcondition is true.

■

6. Implementation

Before we move on to the examples and evaluate the rules introduced in the preceding chapters, we will discuss how they were implemented into the KeY system and its UI.

6.1. Implementation into KeY

While KeY provides a language for *taclets* – a formalization of JavaDL rules which can be loaded at runtime – [RU16], these taclets are not powerful enough to express the rules introduced in this thesis due to the translation from JML to JavaDL and the complex program transformations (see section 3.4.2) involved.

Instead, `blockContractExternal` and `loopContract` were implemented in Java directly in KeY. Their implementation extends the existing implementation of `blockContract`. The user may choose whether they want to apply `blockContract` (which is called "Internal Block Contract" in KeY), `blockContractExternal` (which is called "External Block Contract"), or neither of those rules (in which case the block contract is ignored and the block is simply expanded) when a block is encountered.

As discussed in chapters 7 and 8, `blockContract` is more powerful, since it proves a weaker premiss than `blockContractExternal`, while using `blockContractExternal` leads to shorter proofs.

There is no corresponding internal version for `loopContract`[1]. Instead, we provide two different implementations of this same rule. If "External Block Contract" is selected, only (`pre`) and (`usage`) are proven in the proof for the surrounding method; if "Internal Block Contract" is selected, the (`valid_loop`) branch is also proven in the proof for the surrounding method.

Even though both of these implementations are based on the same rule, and thus equally powerful, using the "internal" version can be a useful time-saving measure if the proof for the (`valid_loop`) branch is not too long.

Loop contracts can also be applied to blocks than start with for-loops. This necessitated the implementation of another rule to transform such a block into one that starts with a while loop. During this transformation, the initializers in the for-loop are moved outside the block, which means that they are not covered by the loop contract. This is slightly unintuitive but necessary as the loop contract rule requires the first element in the block to be the actual loop. It is also consistent with KeY's treatment of loop invariants on for-loops.

[1] More specifically, it is not possible to replace the {anonIn} update with {context}. Such a rule would not be sound because instead of showing that the loop body always preserves the precondition, it would only show that the very first execution of the loop body preserves the precondition.

6.2. Integration into KeY's UI

When symbolic execution reaches a block with a contract in the interactive mode, the user may select that block in the sequent and then choose between the two rules in the context menu which opens (see figure 6.2). If the block has multiple contracts, the user may choose which one(s) to apply. By default, all contracts are combined[2] and applied.

The same is true for loop contracts.

Figure 6.1.: KeY's context menu for rule application

In the automatic mode, KeY always applies the rule the user has selected in the *Proof Search Strategy* tab. There is also an option to ignore all block contracts and expand every block into its surrounding method (figure 6.2).

When a block contract is applied, the proof is split into two or three branches, depending on the rule chosen. Figure 6.3 shows a proof tree in KeY: Every node is labeled with the rule that used to obtain that node. The last node, labeled *Block Contract (Internal)*, splits the proof into three branches.

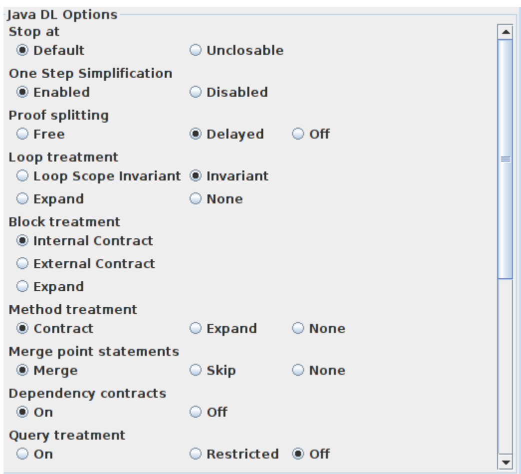

Figure 6.2.: Proof Search Strategy settings

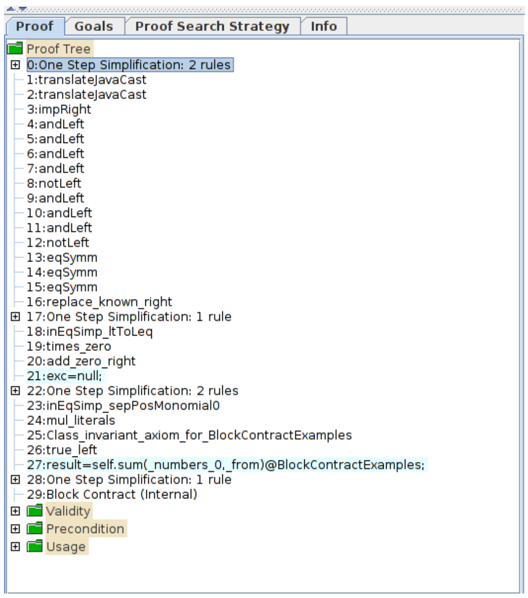

Figure 6.3.: A proof split into three branches

[2]with the combined precondition being the disjunction of all preconditions and the combined postcondition taking the form $\bigwedge_i (\mathrm{pre}_i \rightarrow \mathrm{post}_i)$

The proof for a method is only considered closed if all block contracts used in it have been proven valid as well (figure 6.4).

Until this is the case, the proof for the method is only considered partially closed (figure 6.5).

If a block contract is proven with the internal rule, or the block is expanded, the proof obligation for that contract becomes unnecessary and is grayed out in the selection window (figure 6.4). The grayed out proof obligation may still be selected and proved. This is because even if the user has proved the block contract using the internal rule, they may still want to prove the stronger proposition (`valid*`) (see section 4.1 for the differences between (`valid*`) and (`valid`)).

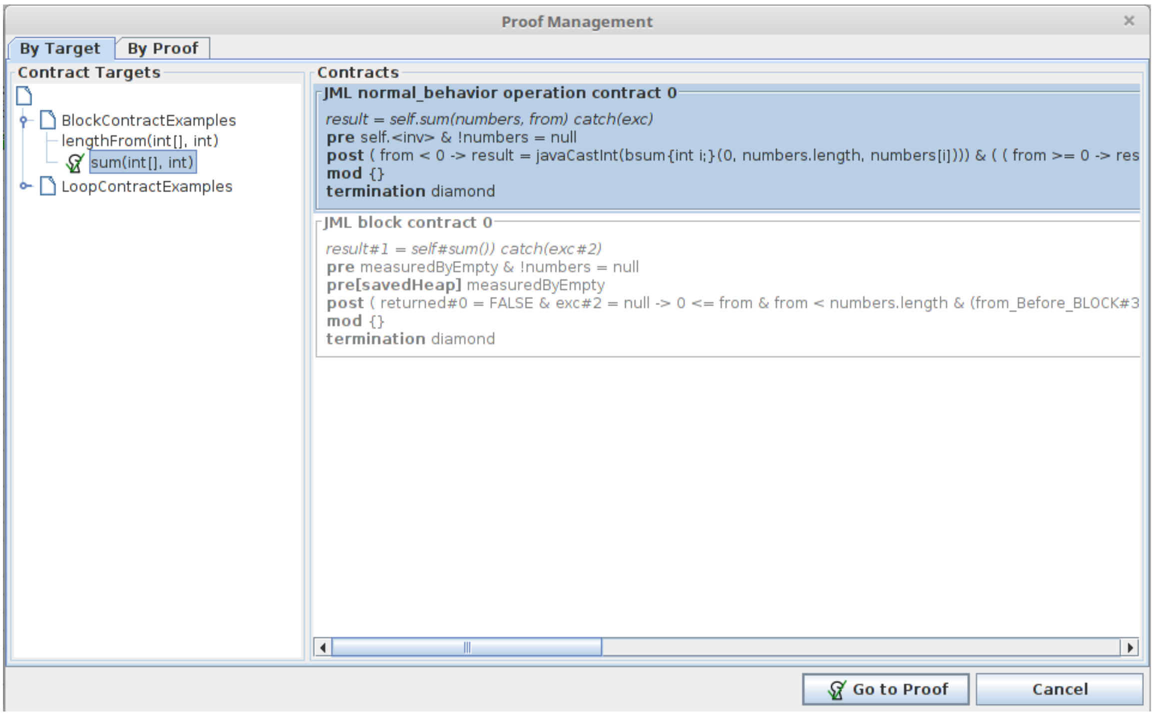

Figure 6.4.: A closed proof in the proof selection window

Figure 6.5.: A partially closed proof

7. Evaluation

In this chapter, we will evaluate the rules introduced in chapters 4 and 5.

In the introduction (section 1.2), we stated that our goal was to introduce a way to divide proofs into smaller, independent subproofs without having to refactor the program. We will test whether `blockContractExternal` accomplishes this, and compare the size a proof for a method divided into blocks to a proof of the same method divided into sub-methods.

We will also test in which cases substituting `blockContract` for `blockContractExternal` offers a performance benefit.

Last but not least, we will evaluate `loopContract` and compare loop contracts to loop invariants.

7.1. Block Contracts

7.1.1. Introductory Example

Let us again consider our introductory block contract example from section 2.2.2:

```
/*@ ... @*/
public int sum(int[] numbers, int from) {
    /*@ requires numbers != null;
      @ ensures 0 <= from && from < numbers.length
      @   && (\old(from) < 0 ==> from == 0)
      @   && (\old(from) >= 0 ==> from == \old(from));
      @ returns \result == 0
      @   && (\old(from) >= numbers.length
      @       || numbers.length == 0);
      @ signals_only \nothing;
      @ assignable \nothing;
      @*/
    {
        if (from < 0) {
            from = 0;
        }

        if (from >= numbers.length) {
            return 0;
        }
    }
    // ...
```

```
}
```

Since `numbers` is a non-nullable argument, the block's precondition `requires numbers != null;` is only necessary when using the rule `blockContractExternal` instead of `blockContract`.

To compare the performance of the two rules, we first let KeY prove the method contract with the rule `blockContract` and without any preconditions on the block contract.

Then we add the precondition to the block contract, and run the proof again, still using `blockContract`.

Then, we prove the method contract using `blockContractExternal`, which requires two proofs, one to prove the block's validity (i.e., the premiss (`valid`*)) and one for to prove the method's validity (the branches (`pre`) and (`usage`)).

We obtain the following results[1]:

Used Rule	Proof Steps	Runtime
Internal (without preconditions)	950	808 ms
Internal (with preconditions)	950	606 ms
External (surrounding method)	734	477 ms
External (block)	250	171 ms
External (total)	984	648 ms

As we can see, `blockContractExternal` does not offer any advantage for such a small example, as total number of proof steps, as well as the total runtime, is larger than when using `blockContract`.

7.1.2. Divide and Conquer with Block Contracts

In this example from [BSSU17], we will show how block contracts can serve as an alternative to splitting a method into sub-methods for easier verification.

To facilitate the proof performed in [BSSU17], many large methods were split into multiple sub-methods. The local variables needed by more than one sub-method were made into attributes. For instance, the method `prepare_indices` shown below was split into two sub-methods `calcE` and `eInsertionSort`:

```
/*@ <prepare_indices contract> @*/
static void prepare_indices(int[] a, int left, int right) {
    calcE(left, right);
    eInsertionSort(a,left,right,e1,e2,e3,e4,e5);
}
```

[1] on a machine with an Intel Core i7-4720HQ (2×2.60GHz) and 16 GB of RAM running Windows 8.1

```
/*@ <calcE contract> @*/
static void calcE(int left, int right) { /* <calcE body> */ }

/*@ <eInsertionSort contract> @*/
static void eInsertionSort(
        int[] a, int left, int right,
        int e1, int e2, int e3, int e4, int e5)
    { /* <eInsertionSort body> */ }
```

`blockContractExternal` allows us to inline the sub-methods while still being able to divide the proof for `prepare_indices` into two sub-proofs[2]:

```
/*@ <prepare_indices contract> @*/
static void prepare_indices(int[] a, int left, int right) {
    /*@ <calcE contract> @*/
    { /* <calcE body> */ }

    /*@ <eInsertionSort contract> @*/
    { /* <eInsertionSort body> */ }
}
```

When proving the validity of `prepare_indices` with the two versions, we get the following results:

	`prepareIndices`	`calcE` (or corresponding block)	`eInsertionSort` (or corresponding block)
Sub-Methods	2358 steps 2522 ms	24533 steps 52962 ms	162348 steps 434111 ms
Blocks	3628 steps 3805 ms	4861 steps 5205 ms	136956 steps 327844 ms

The differences in performance seem to be mostly due to KeY's automatic proof search. For instance, the method `calcE` and its corresponding block contain the exact same code and have the same contract, so there must exist a proof for `calcE` whose size is much closer to 4861 steps. In fact, with some interaction, we can find a proof with only 5086 steps and a runtime (in the automatic mode) of 5527 ms[3].

[2] The full source code for all examples in this chapter can be found in the appendix. Additionally, all examples presented here are included in version 2.7 of KeY under the directory *Dynamic Frames/Block & Loop Contracts*.

[3] This proof, as well as all other non-trivial proofs from this chapter, is also included in version 2.7 of KeY.

Nevertheless, we have shown that using block contracts instead of sub-methods does not impact KeY's performance negatively, while lessening the specification effort. We could go even further and do this for the whole algorithm verified in [BSSU17], which would also allow us to avoid turning any local variables into attributes, thus keeping the verified code reentrant. This is, however, out of the scope of this thesis.

7.1.3. Comparison Between the Block Contract Rules

The purpose of this example, also from [BSSU17], is to illustrate the difference in specification effort necessary to use `blockContract` and `blockContractExternal`.

Consider the method `eInsertionSort` from the previous example. This method is split into 4 blocks, and the validity of its contract can only be proven using `blockContract` because the block contracts contain no preconditions.

In order to be able to use `blockContractExternal` instead of `blockContract`, we need to add additional preconditions to every block.

For the sake of readability, we will introduce some abbreviations. Note that these are not a part of JML's syntax!

```
LR1   := 0 <= left && left < e1
            && e5 < right && right < a.length;
LR2   := left < e1 && e1 < e2 && e2 < e3
            && e3 < e4 && e4 < e5 && e5 < right;
SORT1 := (\forall int i; 0 <= i && i < left;
            (\forall int j; left <= j && j < a.length;
               a[i] <= a[j]));
SORT2 := (\forall int i; 0 <= i && i <= right;
            (\forall int j; right < j && j < a.length;
               a[i] <= a[j]));
```

With these abbreviations, the fully specified method reads as follows. This specification is equivalent to that in [BSSU17] except for the `requires` clauses on the block contracts.

```
/*@ normal_behaviour
  @ requires a.length > 46;
  @ requires LR1 && LR2 && SORT1 && SORT2;
  @ ensures SORT1 && SORT2;
  @ ensures a[e1] <= a[e2] && a[e2] <= a[e3];
  @ ensures a[e3] <= a[e4] && a[e4] <= a[e5];
  @ assignable a[left..right];
  @*/
static void eInsertionSort(
        int[] a, int left, int right,
        int e1, int e2, int e3, int e4, int e5) {
    /*@ requires a != null;
      @ requires LR1 && LR2 && SORT1 && SORT2;
      @ ensures SORT1 && SORT2;
```

```
      @ ensures (a[e1] <= a[e2]);
      @ assignable a[e1], a[e2];
      @ signals_only \nothing;
      @*/
    { /* ... */ }

    /*@ requires a != null;
      @ requires LR1 && LR2 && SORT1 && SORT2;
      @ requires (a[e1] <= a[e2]);
      @ ensures SORT1 && SORT2;
      @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]);
      @ assignable a[e1], a[e2], a[e3];
      @ signals_only \nothing;
      @*/
    { /* ... */ }

    /*@ requires a != null;
      @ requires LR1 && LR2 && SORT1 && SORT2;
      @ requires (a[e1] <= a[e2] && a[e2] <= a[e3]);
      @ ensures SORT1 && SORT2;
      @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]
      @       && a[e3] <= a[e4]);
      @ assignable a[e1], a[e2], a[e3], a[e4];
      @ signals_only \nothing;
      @*/
    { /* ... */ }

    /*@ requires a != null;
      @ requires LR1 && LR2 && SORT1 && SORT2;
      @ requires (a[e1] <= a[e2] && a[e2] <= a[e3]
      @        && a[e3] <= a[e4]);
      @ ensures SORT1 && SORT2;
      @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]
      @       && a[e3] <= a[e4] && a[e4] <= a[e5]);
      @ assignable a[e1], a[e2], a[e3], a[e4], a[e5];
      @ signals_only \nothing;
      @*/
    { /* ... */ }
}
```

As we can see, using `blockContractExternal` instead of `blockContract` requires a larger specification effort. It does, however, improve performance.

Once again, we let KeY prove the method contract using the rule `blockContract`, with and without any preconditions on the block contracts. Then, we prove the method contract using `blockContractExternal`.

Used Rule	Proof Steps	Runtime
Internal (without preconditions)	162348	434111 ms
Internal (with preconditions)	153037	348061 ms
External (surrounding method)	31559	74942 ms
External (blocks)	4540 + 12246 + 29026 + 55724	(2585 + 8322 + 25158 + 61764) ms
External (total)	133095	172771 ms

As we can see, the additional preconditions already slightly improve KeY's performance even without `blockContractExternal`. Using `blockContractExternal` improves the performance even further.

7.2. Loop Contracts

The purpose of the following two examples is to show how `loopContract` can be used to divide a proof into two sub-proofs. We also compare the size of these divided proofs with that of equivalent proofs which use KeY's loop invariant rule instead.

7.2.1. Array Increment

Consider the following example, adapted from [Tue12].

```
/*@ normal_behavior
  @ requires arr != null;
  @ ensures (\forall int i; 0 <= i && i < arr.length;
  @       arr[i] == \old(arr[i]) + 1);
  @ assignable arr[*];
  @*/
public static void mapIncrement(int[] arr) {
    int i = 0;
    while (i < arr.length) {
        ++arr[i];
        ++i;
    }
}
```

We can specify the loop in this method with a loop contract:

```
/*@ loop_contract normal_behavior
  @ requires arr != null && 0 <= i && i <= arr.length;
  @ ensures (\forall int j;
```

```
  @          \before(i) <= j && j < arr.length;
  @          arr[j] == \before(arr[j]) + 1);
  @ assignable arr[i .. arr.length];
  @ decreases arr.length - i;
  @*/
```

or with a loop invariant:

```
/*@ loop_invariant (0 <= i && i <= arr.length);
  @ loop_invariant (\forall int j; 0 <= j && j < i;
  @      arr[j] == \old(arr[j]) + 1);
  @ loop_invariant (\forall int j; i <= j && j < arr.length;
  @      arr[j] == \old(arr[j]));
  @ assignable arr[i .. arr.length];
  @ decreases arr.length - i;
  @*/
```

Before we continue with the performance comparison, let us note the differences between the loop contract and the equivalent loop invariant: The loop invariant states which array elements have already been incremented and which have not. The loop contract is more intuitive, stating instead which elements are still going to be incremented and which are not. Which elements will not be incremented is stated via an `assignable` clause instead of a universal quantifier, making the loop contract slightly easier to read. In fact, the `assignable` clause on the invariant is superfluous and only stated here for completeness' sake.

We let KeY prove both versions of the method. The version with a loop contract requires two proofs whose number of steps and runtime we add.

Proof	Proof Steps	Runtime
Loop Contract (surrounding method)	154	108 ms
Loop Contract (loop)	1432	2109 ms
Loop Contract (total)	1586	2217 ms
Loop Invariant	1087	705 ms

As we can see, the two versions have comparable performance, with the invariant one being slightly better.

7.2.2. List Increment

We will now convert the above example to linked lists.

For this, we will use a JML feature called *ghost variables*. A ghost variable is a variable that can not be accessed by the Java code, but only by the JML specification. We introduce two ghost attributes of type `\seq` to our linked list class, `\seq` being a JML and JavaDL type for finite sequences [SB16, 2]. The first ghost attribute `seq` is the sequence of all dates in our list, while the second attribute `nodeseq` is the sequence of all nodes.

Our linked list also has a *class invariant*. A class invariant is a formula that is automatically added to the pre- and postcondition of every method in the class [HAGH16, 4.1].

The class invariants are taken on an existing KeY example written by Mattias Ulbrich[4].

```
public final class IntNode {
    public /*@ nullable @*/ IntNode next;
    public int data;
}
```

```
public interface IntList {
    /*@ public ghost \seq seq; */
    /*@ invariant ...; */
    // ...
}
```

```
public final class IntLinkedList implements IntList {

    /*@ nullable @*/ IntNode first;
    /*@ nullable @*/ IntNode last;
    int size;

    /*@ ghost \seq nodeseq; */

    /*@ normal_behavior
      @ ensures (\forall int i; 0 <= i && i < size;
      @      ((int) seq[i]) == \old((int) seq[i]) + 1);
      @ ensures size == \old(size);
      @ assignable \set_union(\singleton(seq),
      @      \infinite_union(int j; 0 <= j && j < size;
      @          \singleton(((IntNode)nodeseq[j]).data)));
      @*/
    public void mapIncrement() {
        IntNode current = first;
        int i = 0;
        while (current != null) {
            ++current.data;
```

[4]This example can be accessed under the directory *Dynamic Frames/List with Sequences*

```
            // The following statement inserts the
            // incremented value into seq.
            /*@ set seq =
              @     \seq_concat(\seq_sub(seq, 0, i),
              @         \seq_concat(
              @             \seq_singleton(current.data),
              @             \seq_sub(seq, i+1, size)));
              @*/

            current = current.next;
            ++i;
        }
    }
}
```

The method's `assignable` clause makes use of some JML set operations; it states that the method may modify the ghost variable `seq` as well as the `data` attribute of every node contained in `nodeseq`.

Again, we can specify the loop in this method using a loop contract:

```
/*@ loop_contract normal_behavior
  @ requires \invariant_for(this);
  @ requires 0 <= i && i <= size;
  @ requires i < size
  @     ==> current == (IntNode) nodeseq[i];
  @ requires i == size ==> current == null;
  @ ensures \invariant_for(this);
  @ ensures (\forall int j; \before(i) <= j && j < size;
  @     (int) seq[j] == \before((int) seq[j]) + 1);
  @ ensures size == \before(size);
  @ assignable \set_union(\singleton(seq),
  @     \infinite_union(int j; i <= j && j < size;
  @         \singleton(((IntNode)nodeseq[j]).data)));
  @ decreases nodeseq.length - i;
  @*/
```

or a loop invariant:

```
/*@ loop_invariant \invariant_for(this);
  @ loop_invariant 0 <= i && i <= size;
  @ loop_invariant i < size
  @     ==> current == (IntNode) nodeseq[i];
  @ loop_invariant i == size ==> current == null;
  @ loop_invariant (\forall int j; 0 <= j && j < i;
  @     (int) seq[j] == \old((int) seq[j]) + 1);
  @ loop_invariant (\forall int j; i <= j && j < size;
```

```
@     (int) seq[j] == \old((int) seq[j]));
@ loop_invariant size == \old(size);
@ assignable \set_union(\singleton(seq),
@     \infinite_union(int j; i <= j && j < size;
@         \singleton(((IntNode)nodeseq[j]).data)));
@ decreases nodeseq.length - i;
@*/
```

Note the same differences between the loop contract and the loop invariant as before: The loop invariant states which array elements have already been incremented and which have not. The loop contract states which elements are still going to be incremented and which are not. Which elements will not be incremented is again stated via an `assignable` clause instead of a universal quantifier, and again, the `assignable` clause on the invariant is superfluous and only stated for completeness' sake.

The performance difference is as follows:

Proof	Automatic Proof Steps	Interactive Proof Steps	Runtime in Automatic Mode
Loop Contract (surrounding method)	21521	0	29107 ms
Loop Contract (loop)	56560	70	623884 ms
Loop Contract (total)	78081	70	652991 ms
Loop Invariant	79389	63	842628 ms

The number of interactive proof steps in the above table is somewhat misleading, as the proof for the loop contract only requires user interaction in one branch, while the proof for the loop invariant requires user interaction in two branches.

Specifically, in both cases proving

```
(\forall int j; ...; (int) seq[j] == \old((int) seq[j]) + 1)
```

or

```
(\forall int j; ...; (int) seq[j] == \before((int) seq[j]) + 1)
```

respectively requires interaction: After the proof is manually split into the two cases `j == i` and `j != i`, the branch for `j != i` closes automatically, while the branch for `j == i` requires further interaction (where `i` is the current value of the index variable).

The loop invariant also requires interaction to show a part of the class invariant, namely `nodesec[i].data = sec[i]`.

As we can see, `loopContract` has the same advantages as `blockContractExternal` in that it allows us to divide a proof into two sub-proofs without increasing the divided proof's size and thus reducing KeY's performance.

Furthermore, we have shown that there are cases in which a loop contract is easier to specify and prove than a loop invariant.

8. Conclusions

In the preceding chapters, we introduced two new rules, `blockContractExternal` and `loopContract`, for the application of block and loop contracts in the sequent calculus for JavaDL.

These rules allow us to prove a block's correctness in a separate proof obligation, thus allowing us to divide a proof over a complex method into sub-proofs without having to actually divide the method into multiple sub-methods.

8.1. Results

When comparing the size of two proofs for the same method, one where the method was divided into two blocks and one where it was divided into two sub-methods, we found that dividing the method into blocks instead of sub-methods lowers the specification effort (as we have to perform less refactoring) without increasing the proof's size or KeY's runtime.

When comparing `blockContractExternal` to `blockContract`, we found that using `blockContractExternal` requires a larger specification effort because the separation of the block validity proof necessitates that the block contract be universally valid (instead of only valid in the context in which it occurs). However, we found that for complex enough methods, this specification effort leads to a smaller overall proof size regardless of which block contract rule we use; though using `blockContractExternal` leads to the smallest proof.

We also compared loop contracts to loop invariants and found that loop contracts require a similar specification effort to loop invariants, actually being more readable in certain situations while offering approximately the same performance with the additional advantage that `loopContract` allows us to divide a proof.

8.2. Outlook

The rules introduced in this thesis have been implemented into KeY. Some aspects of this implementation could still be improved. Firstly, we introduced loop contracts as special block contracts for blocks that start with a loop. However, none of the examples we looked at had any code after the loop. For this reason, allowing the user to put a loop contract directly on a loop instead of a surrounding block might be a useful addition. Secondly, well-definedness checks for loop contracts have not been implemented, which means that ill-defined loop contracts cannot be detected by KeY.

Instead of using the concept of loop contracts, we could also have implemented an alternative rule for loop invariants in which the *Body preserves invariant* branch is a separate proof obligation. While we have demonstrated that loop contracts are easier to specify and read in some situations, there may be situations in which one would rather use a loop invariant while still being able to divide the resulting proof.

Lastly, this thesis considered only proofs for functional correctness in Java. Obviously, this divide-and-conquer strategy is applicable to other languages as well. Furthermore, one could investigate whether a similar strategy can be applied to information flow proofs.

Bibliography

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[BKW16] Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. Dynamic Logic for Java. In Ahrendt et al. [ABB+16], chapter 3.

[BSSU17] Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt, and Mattias Ulbrich. Proving JDK's dual pivot quicksort correct. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2017)*, July 2017.

[GBM+16] Daniel Grahl, Richard Bubel, Wojciech Mostowski, Peter H. Schmitt, Mattias Ulbrich, and Benjamin Weiß. Modular Specification and Verification. In Ahrendt et al. [ABB+16], chapter 9.

[GU16] Daniel Grahl and Mattias Ulbrich. From Specification to Proof Obligations. In Ahrendt et al. [ABB+16], chapter 8.

[HAGH16] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. Formal Specification with the Java Modeling Language. In Ahrendt et al. [ABB+16], chapter 7.

[LPC+13] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. *JML Reference Manual*. May 2013.

[RU16] Philipp Rümmer and Mattias Ulbrich. Proof Search with Taclets. In Ahrendt et al. [ABB+16], chapter 4.

[SB16] Peter H. Schmitt and Richard Bubel. Theories. In Ahrendt et al. [ABB+16], chapter 5.

[Sch16] Peter H. Schmitt. First-Order Logic. In Ahrendt et al. [ABB+16], chapter 2.

[Tue12] Thomas Tuerk. Local reasoning about while-loops. In *International Conference on Verified Software: Theories, Tools and Experiments - Theory Workshop (VS-Theory)*, May 2012.

[Wac12] Simon Wacker. Blockverträge. Studienarbeit, Karlsruher Institut für Technologie, October 2012.

A. Soundness Proof for the Loop Contract Rule

This appendix contains the full soundness proof for the rule `loopContract` defined in section 5.6. A proof sketch can be found in section 5.7.

Let `block`′ be the transformed form of

$$\{\texttt{while}(\text{loopCond})\{\text{body}\}\texttt{tail}\}$$

as described in 3.4.2, i.e., the program fragment

```
method-frame(this=self) : {
    boolean broke = false;
    boolean broke₁ = false;
    ⋮
    boolean broke_ξ = false;

    boolean continued = false;
    boolean continued₁ = false;
    ⋮
    boolean continued_ξ = false;

    boolean returned = false;

    Throwable exception = null;

    breakOut: try {
        breakLoop: while (loopCondition) { body^almostSafe }
        tail^almostSafe
    } catch (Throwable e) {
        exception = e;
    }
}
```

and `loop` the following:

```
method-frame(this=self) : {
    boolean broke = false;
    boolean brokeLoop = false;
```

```
        boolean broke_1 = false;
        ⋮
        boolean broke_ξ = false;

        boolean continued_1 = false;
        ⋮
        boolean continued_ξ = false;

        boolean returned = false;

        Throwable exception = null;

        breakOut: try {
            breakLoop: while (loopCondition) { body^almostSafe }
        } catch (Throwable e) {
            exception = e;
        }
    }
```

We will now prove that the validity of (`valid_loop`) implies the validity of (`valid`*), i.e.,

$$\frac{(\texttt{valid\_loop})}{\begin{array}{l}\mathrm{wellFormed}(\texttt{heap}), \mathrm{wellFormed}(\texttt{heap}^{\mathrm{anon}}) \\ \quad \Longrightarrow \{\mathrm{rememberOuter}\}\{\mathrm{anonIn}\}(\mathrm{pre} \wedge \mathrm{wellFormed}(\texttt{heap}) \wedge \mathrm{reachableIn} \\ \qquad \rightarrow \{\mathrm{remember}\}[\![\texttt{block}']\!](\mathrm{post} \wedge \mathrm{frame})\end{array}}$$

Let $\Sigma = (\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym}, \mathrm{ProgVSym})$ be a JavaDL signature.

If $\not\models$ `valid_loop`, we are done.

If $\models$ `valid_loop`, we need to prove that $\models$ `valid`*.

Let $K = (S, \rho)$ be a Kripke structure for Σ, $s \in S$ a state, $\beta : \mathrm{VSym} \to D$ a variable assignment, and $s_a = \mathrm{val}_{(K,s,\beta)}(\{\{\mathrm{rememberOuter}\}\mathrm{anonIn}\})(s)$

Case 1: $(K, s, \beta) \not\models \mathrm{wellFormed}(\texttt{heap}) \wedge \mathrm{wellFormed}(\texttt{heap}^{\mathrm{anon}})$ $\wedge\{\mathrm{rememberOuter}\}\{\mathrm{anonIn}\}(\mathrm{pre} \wedge \mathrm{wellFormed}(\texttt{heap}) \wedge \mathrm{reachableIn})$

Trivial.

Case 2: $(K, s_a, \beta) \models \{\mathrm{remember}_{\mathrm{current}}\}[\![\texttt{unfold}']\!]\texttt{exception} \dot{\neq} \texttt{null}$

Case 3: $(K, s_a, \beta) \models \{\mathrm{remember}_{\mathrm{current}}\}[\![\texttt{unfold}']\!](\texttt{exception} \doteq \texttt{null} \wedge \texttt{cond} \doteq \mathrm{FALSE})$

Case 4:

$$
\begin{aligned}
(K, s_a, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!](\\
&\qquad\qquad \texttt{exception} \doteq \texttt{null} \wedge \texttt{cond} \doteq \texttt{TRUE} \wedge [\![\text{body}']\!]\texttt{brokeLoop} \doteq \texttt{TRUE})
\end{aligned}
$$

Case 5:

$$
\begin{aligned}
(K, s_a, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!](\\
&\qquad\qquad \texttt{exception} \doteq \texttt{null} \wedge \texttt{cond} \doteq \texttt{TRUE} \wedge [\![\text{body}']\!]\text{abrupt})
\end{aligned}
$$

In the cases 2 to 5, we can conclude from the definitions of our program fragments that $(K, s_a, \beta) \models \{\text{remember}_{\text{current}}\}[\![\texttt{block}']\!](\text{post}_{\text{current}} \wedge \text{frame}_{\text{current}})$, and thus $(K, s, \beta) \models \texttt{valid}^*$.

Case 6:

$$
\begin{aligned}
(K, s_a, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!](\\
&\qquad\qquad \texttt{exception} \doteq \texttt{null} \wedge \texttt{cond} \doteq \texttt{TRUE} \wedge [\![\text{body}']\!](\\
&\qquad\qquad\qquad \texttt{brokeLoop} \doteq \texttt{FALSE} \wedge \neg\text{abrupt}))
\end{aligned}
$$

Case 6.1: If $(K, s_a, \beta) \models \texttt{valid\_loop}$ because one of the program fragments does not terminate (and $[\![]\!] = []$), $\texttt{block}'$ does not terminate either, and we are done (If $[\![]\!] = \langle\rangle$, then $\models \texttt{valid\_loop}$ implies that the program fragments always terminate).

Case 6.2: If $\texttt{unfold}'$, $\texttt{body}'$ and $\texttt{tail}'$ terminate, $\texttt{block}'$ also terminates because of the decreasesCheck.

We now define the following abbreviations:

$$
\begin{aligned}
\text{assumptions} =& \text{wellFormed}(\texttt{heap}^{\text{pre}}) \wedge \text{wellFormed}(\texttt{heap}^{\text{anon}})\\
& \wedge \text{pre} \wedge \text{decreasesCheck}\\
\text{conditions}_{\text{subscr}} =& \text{post}_{\text{subscr}} \wedge \text{frame}_{\text{subscr}}\\
& \text{for subscr} \in \{\text{current}, \text{next}\}
\end{aligned}
$$

We know that

$$
\begin{aligned}
(K, s_a, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\text{body}']\!](\text{assumptions}\\
&\qquad \wedge \{\text{remember}_{\text{next}}\}\{\text{anonOut}_{\texttt{loop}}\}\\
&\qquad\qquad (\text{abrupt} \rightarrow (\text{conditions}_{\text{next}} \rightarrow \text{conditions}_{\text{current}})\\
&\qquad\qquad \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!](\text{conditions}_{\text{next}} \rightarrow \text{conditions}_{\text{current}}))))
\end{aligned}
$$

Now let $\mathrm{val}_{(K,s,\beta)}(v_i^{\mathrm{anon}})$ be the value of v_i after the execution of $[\![\texttt{unfold}']\!]$, $[\![\mathrm{body}']\!]$, and $[\![\texttt{loop}]\!]$. Then

$$\begin{aligned}(K, s_a, \beta) \models &\{\mathrm{remember}_{\mathrm{current}}\}[\![\texttt{unfold}']\!][\![\mathrm{body}']\!](\mathrm{assumptions}\\ &\quad \wedge \{\mathrm{remember}_{\mathrm{next}}\}[\![\texttt{loop}]\!]\\ &\qquad (\mathrm{abrupt} \rightarrow (\mathrm{conditions}_{\mathrm{next}} \rightarrow \mathrm{conditions}_{\mathrm{current}})\\ &\qquad \wedge (\neg\mathrm{abrupt} \rightarrow [\![\texttt{tail}']\!](\mathrm{conditions}_{\mathrm{next}} \rightarrow \mathrm{conditions}_{\mathrm{current}}))))\end{aligned}$$

Case 6.2.1: After the the last loop iteration, $\texttt{brokeLoop}\doteq\mathrm{TRUE} \vee \mathrm{abrupt}$.

In this case, we know that there exists a $n \in \mathbb{N}$ so that the above is equivalent to

$$\begin{aligned}(K, s_a, \beta) \models &\{\mathrm{remember}_{\mathrm{current}}\}[\![\texttt{unfold}']\!][\![\mathrm{body}']\!](\mathrm{assumptions}\\ &\quad \wedge \{\mathrm{remember}_{\mathrm{next}}\}\underbrace{[\![\texttt{unfold}']\!][\![\mathrm{body}']\!]\ldots[\![\texttt{unfold}']\!][\![\mathrm{body}']\!]}_{(n-1)\text{ times}}\\ &\qquad (\mathrm{abrupt} \rightarrow (\mathrm{conditions}_{\mathrm{next}} \rightarrow \mathrm{conditions}_{\mathrm{current}})\\ &\qquad \wedge (\neg\mathrm{abrupt} \rightarrow [\![\texttt{tail}']\!](\mathrm{conditions}_{\mathrm{next}} \rightarrow \mathrm{conditions}_{\mathrm{current}}))))\end{aligned}$$

Because $\models$ `valid_loop`, we know that the assumptions are preserved between all n iterations.

Now let s^n be the state before the last iteration and s^{n-1} the state before the second-to-last iteration.

In other words, s^i results from s by executing $[\![\texttt{unfold}']\!][\![\mathrm{body}']\!]$ $(i-1)$ times.

Then (K, s^n, β) conforms to **Case 4** or **Case 5**, and thus

$$(K, s_a^n, \beta) \models \{\mathrm{remember}_{\mathrm{current}}\}[\![\texttt{unfold}']\!][\![\mathrm{body}']\!]\mathrm{conditions}_{\mathrm{current}}$$

which is equivalent to

$$(K, s_a^n, \beta) \models \{\mathrm{remember}_{\mathrm{next}}\}[\![\texttt{unfold}']\!][\![\mathrm{body}']\!]\mathrm{conditions}_{\mathrm{next}}$$

This means that

$$
\begin{aligned}
(K, s_a^{n-1}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\text{body}']\!](\text{assumptions} \\
&\quad \wedge \{\text{remember}_{\text{next}}\}[\![\texttt{unfold}']\!][\![\text{body}']\!] \\
&\qquad (\text{abrupt} \rightarrow \text{conditions}_{\text{next}} \\
&\qquad\ \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!]\text{conditions}_{\text{next}})))
\end{aligned}
$$

and thus

$$
\begin{aligned}
(K, s_a^{n-1}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\text{body}']\!](\text{assumptions} \\
&\quad \wedge \{\text{remember}_{\text{next}}\}[\![\texttt{unfold}']\!][\![\text{body}']\!] \\
&\qquad (\text{abrupt} \rightarrow \text{conditions}_{\text{current}} \\
&\qquad\ \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!]\text{conditions}_{\text{current}})))
\end{aligned}
$$

Now let s^i, s^{i-1} be the states before the ith and $(i-1)$th iteration respectively. We assume that $(K, s_a^i, \beta) \models \{\text{remember}_{\text{current}}\}[\![\texttt{block}']\!]\text{post}_{\text{current}}$.

Because there are $n \geq i$ iterations, we know that

$$
\begin{aligned}
(K, s_a^{i-1}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!](\texttt{exception} \doteq \texttt{null} \wedge \texttt{cond} \doteq \texttt{TRUE} \\
&\quad \wedge [\![\text{body}']\!](\texttt{brokeLoop} \doteq \texttt{FALSE} \wedge \neg\text{abrupt}))
\end{aligned}
$$

and thus

$$
\begin{aligned}
(K, s_a^{i-1}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\text{body}']\!](\text{assumptions} \\
&\quad \wedge \{\text{remember}_{\text{next}}\} \underbrace{[\![\texttt{unfold}']\!][\![\text{body}']\!] \ldots [\![\texttt{unfold}']\!][\![\text{body}']\!]}_{(n-i+1) \text{ times}} \\
&\qquad (\text{abrupt} \rightarrow (\text{conditions}_{\text{next}} \rightarrow \text{conditions}_{\text{current}}) \\
&\qquad\ \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!](\text{conditions}_{\text{next}} \rightarrow \text{conditions}_{\text{current}}))))
\end{aligned}
$$

From our induction hypothesis, we can conclude

$$
\begin{aligned}
(K, s_a^{i-1}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\text{body}']\!](\text{assumptions} \\
&\quad \wedge \{\text{remember}_{\text{next}}\} \underbrace{[\![\texttt{unfold}']\!][\![\text{body}']\!] \ldots [\![\texttt{unfold}']\!][\![\text{body}']\!]}_{(n-i+1) \text{ times}} \\
&\qquad (\text{abrupt} \rightarrow \text{conditions}_{\text{next}} \\
&\qquad\ \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!]\text{conditions}_{\text{next}})))
\end{aligned}
$$

which implies

$$
\begin{aligned}
(K, s_a^{i-1}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\texttt{body}']\!](\text{assumptions} \\
&\wedge \{\text{remember}_{\text{next}}\} \underbrace{[\![\texttt{unfold}']\!][\![\texttt{body}']\!] \ldots [\![\texttt{unfold}']\!][\![\texttt{body}']\!]}_{(n-i+1)\text{ times}} \\
&\quad (\text{abrupt} \rightarrow \text{conditions}_{\text{current}} \\
&\quad \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!]\text{conditions}_{\text{current}})))
\end{aligned}
$$

Altogether, we now know that

$$
\begin{aligned}
(K, s_a, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\texttt{body}']\!](\text{assumptions} \\
&\wedge \{\text{remember}_{\text{next}}\}[\![\texttt{loop}']\!] \\
&\quad (\text{abrupt} \rightarrow \text{conditions}_{\text{current}} \\
&\quad \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!]\text{conditions}_{\text{current}})))
\end{aligned}
$$

which directly implies $(K, s_a^1 = s_a, \beta) \models \{\text{remember}_{\text{current}}\}[\![\texttt{block}']\!]\text{conditions}_{\text{current}}$.
Thus, $(K, s, \beta) \models \texttt{valid}^*$. Because the only thing we have restricted about (K, s, β) is the interpretation of the anonymization constants v_i^{anon}, which do not occur in $\texttt{valid}^*$, this implies $\models \texttt{valid}^*$.

Case 6.2.2: After the the last loop iteration, $\texttt{brokeLoop} \doteq \texttt{FALSE} \wedge \neg\text{abrupt}$
In this case, we know that there exists a $n \in \mathbb{N}$ so that the above is equivalent to

$$
\begin{aligned}
(K, s_a, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\texttt{body}']\!](\text{assumptions} \\
&\wedge \{\text{remember}_{\text{next}}\} \underbrace{[\![\texttt{unfold}']\!][\![\texttt{body}']\!] \ldots [\![\texttt{unfold}']\!][\![\texttt{body}']\!]}_{(n-1)\text{ times}}[\![\texttt{unfold}']\!] \\
&\quad (\text{abrupt} \rightarrow (\text{conditions}_{\text{next}} \rightarrow \text{conditions}_{\text{current}}) \\
&\quad \wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!](\text{conditions}_{\text{next}} \rightarrow \text{conditions}_{\text{current}}))))
\end{aligned}
$$

Because $\models \texttt{valid\_loop}$, we know that the assumptions are preserved between all n iterations.

Now let s^{n+1} be the state after the last loop iteration.
Then (K, s^{n+1}, β) conforms to **Case 1** or **Case 2**, and thus

$$(K, s_a^{n+1}, \beta) \models \{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!]\text{conditions}_{\text{current}}$$

This means that

$$\begin{aligned}(K, s_a^{n}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\texttt{body}']\!](\text{assumptions}\\ &\wedge \{\text{remember}_{\text{next}}\}[\![\texttt{unfold}']\!]\\ &\quad(\text{abrupt} \rightarrow \text{conditions}_{\text{next}}\\ &\quad\wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!]\text{conditions}_{\text{next}})))\end{aligned}$$

and thus

$$\begin{aligned}(K, s_a^{n}, \beta) \models &\{\text{remember}_{\text{current}}\}[\![\texttt{unfold}']\!][\![\texttt{body}']\!](\text{assumptions}\\ &\wedge \{\text{remember}_{\text{next}}\}[\![\texttt{unfold}']\!]\\ &\quad(\text{abrupt} \rightarrow \text{conditions}_{\text{current}}\\ &\quad\wedge (\neg\text{abrupt} \rightarrow [\![\texttt{tail}']\!]\text{conditions}_{\text{current}})))\end{aligned}$$

By proceeding as in **Case 6.2.1**, we can show that

$$(K, s_a^{1} = s_a, \beta) \models \{\text{remember}_{\text{current}}\}[\![\texttt{block}']\!]\text{conditions}_{\text{current}}$$

■

B. Source Code for the Examples

This appendix contains the full source code for the examples from chapter 7. Those examples whose full source is already included in chapter 7 are not repeated here. All examples are also included in version 2.7 of KeY.

B.1. Block Contracts

B.1.1. Divide and Conquer

See 7.1.2.

```
public class DualPivotQuicksort_sort_methods {

    static int less, great;
    static int e1,e2,e3,e4,e5;

    /*@ normal_behaviour
      @ requires 0 <= left && left < right
      @        && right - left >= 46 && right < a.length;
      @ requires a.length > 46;
      @ requires (\forall int i; 0 <= i && i < left;
      @      (\forall int j; left <= j && j < a.length;
      @           a[i] <= a[j]));
      @ requires (\forall int i; 0 <= i && i <= right;
      @      (\forall int j; right < j && j < a.length;
      @           a[i] <= a[j]));
      @ ensures a[e1] <= a[e2] && a[e2] <= a[e3]
      @       && a[e3] <= a[e4] && a[e4] <= a[e5];
      @ ensures left < e1 && e1 < e2 && e2 < e3
      @       && e3 < e4 && e4 < e5 && e5 < right;
      @ ensures (\forall int i; 0 <= i && i < left;
      @      (\forall int j; left <= j && j < a.length;
      @           a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right;
      @      (\forall int j; right < j && j < a.length;
      @           a[i] <= a[j]));
      @ assignable e1,e2,e3,e4,e5, a[left..right];
      @*/
    static void prepare_indices(int[] a, int left, int right) {
        {calcE(left, right);}
```

```
        eInsertionSort(a,left,right,e1,e2,e3,e4,e5);
    }

    /*@
      @ normal_behaviour
      @ requires 0 <= left && left < right && right - left >= 46;
      @ ensures left < e1 && e1 < e2 && e2 < e3
      @         && e3 < e4 && e4 < e5 && e5 < right;
      @ assignable e1,e2,e3,e4,e5;
      @*/
    static void calcE(int left, int right) {
        int length = right - left + 1;
        int seventh = (length / 8) + (length / 64) + 1;
        e3 = (left + right) / 2; // The midpoint
        e2 = e3 - seventh;
        e1 = e2 - seventh;
        e4 = e3 + seventh;
        e5 = e4 + seventh;
    }

    /*@
      @ normal_behaviour
      @ requires a.length > 46;
      @ requires 0 <= left && left < e1
      @          && e5 < right && right < a.length;
      @ requires left < e1 && e1 < e2 && e2 < e3
      @          && e3 < e4 && e4 < e5 && e5 < right;
      @ requires (\forall int i; 0 <= i && i < left;
      @     (\forall int j; left <= j && j < a.length;
      @         a[i] <= a[j]));
      @ requires (\forall int i; 0 <= i && i <= right;
      @     (\forall int j; right < j && j < a.length;
      @         a[i] <= a[j]));
      @ ensures a[e1] <= a[e2] && a[e2] <= a[e3]
      @         && a[e3] <= a[e4] && a[e4] <= a[e5];
      @ ensures (\forall int i; 0 <= i && i < left;
      @     (\forall int j; left <= j && j < a.length;
      @         a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right;
      @     (\forall int j; right < j && j < a.length;
      @         a[i] <= a[j]));
      @ assignable a[left..right];
      @*/
    static void eInsertionSort(
            int[] a, int left, int right,
            int e1, int e2, int e3, int e4, int e5) {
        /*@
```

```
  @ ensures (a[e1] <= a[e2]);
  @ ensures (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @         a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @         a[i] <= a[j]));
  @ assignable a[e1], a[e2];
  @ signals_only \nothing;
  @*/
{
    if (a[e2] < a[e1]) { int t = a[e2]; a[e2] = a[e1]; a[
       e1] = t; }
}

/*@
  @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]);
  @ ensures (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @         a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @         a[i] <= a[j]));
  @ assignable a[e1], a[e2], a[e3];
  @ signals_only \nothing;
  @*/
{
    if (a[e3] < a[e2]) { int t = a[e3]; a[e3] = a[e2]; a[
       e2] = t;
    if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
    }}

/*@
  @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3] && a[e3] <=
       a[e4]);
  @ ensures (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @         a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @         a[i] <= a[j]));
  @ assignable a[e1], a[e2], a[e3], a[e4];
  @ signals_only \nothing;
  @*/
{
    if (a[e4] < a[e3]) { int t = a[e4]; a[e4] = a[e3]; a[
       e3] = t;
```

```
        if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
        if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
        }
        }}

    /*@
      @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3] && a[e3] <=
          a[e4] && a[e4] <= a[e5]);
      @ ensures (\forall int i; 0 <= i && i < left;
      @     (\forall int j; left <= j && j < a.length;
      @         a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right;
      @     (\forall int j; right < j && j < a.length;
      @         a[i] <= a[j]));
      @ assignable a[e1], a[e2], a[e3], a[e4], a[e5];
      @ signals_only \nothing;
      @*/
    {
        if (a[e5] < a[e4]) { int t = a[e5]; a[e5] = a[e4]; a[
            e4] = t;
        if (t < a[e3]) { a[e4] = a[e3]; a[e3] = t;
        if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
        if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
        }
        }
        }}
  }
}
```

Listing B.1: DualPivotQuicksort_sort_methods.java

```
public class DualPivotQuicksort_sort_blocks {

    static int less, great;
    static int e1,e2,e3,e4,e5;

    /*@ normal_behaviour
      @ requires 0 <= left && left < right
      @       && right - left >= 46 && right < a.length;
      @ requires a.length > 46;
      @ requires (\forall int i; 0 <= i && i < left;
      @     (\forall int j; left <= j && j < a.length;
      @         a[i] <= a[j]));
      @ requires (\forall int i; 0 <= i && i <= right;
      @     (\forall int j; right < j && j < a.length;
      @         a[i] <= a[j]));
      @ ensures a[e1] <= a[e2] && a[e2] <= a[e3]
```

```
  @      && a[e3] <= a[e4] && a[e4] <= a[e5];
  @ ensures left < e1 && e1 < e2 && e2 < e3
  @      && e3 < e4 && e4 < e5 && e5 < right;
  @ ensures (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @         a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @         a[i] <= a[j]));
  @ assignable e1,e2,e3,e4,e5, a[left..right];
  @*/
static void prepare_indices(int[] a, int left, int right) {
    /*@
      @ normal_behaviour
      @ requires 0 <= left && left < right && right - left >=
          46;
      @ ensures left < e1 && e1 < e2 && e2 < e3
      @      && e3 < e4 && e4 < e5 && e5 < right;
      @ assignable e1,e2,e3,e4,e5;
      @*/
    {
        int length = right - left + 1;
        int seventh = (length / 8) + (length / 64) + 1;
        e3 = (left + right) / 2; // The midpoint
        e2 = e3 - seventh;
        e1 = e2 - seventh;
        e4 = e3 + seventh;
        e5 = e4 + seventh;
    }

    /*@
      @ normal_behaviour
      @ requires a.length > 46;
      @ requires 0 <= left && left < e1
      @         && e5 < right && right < a.length;
      @ requires left < e1 && e1 < e2 && e2 < e3
      @         && e3 < e4 && e4 < e5 && e5 < right;
      @ requires (\forall int i; 0 <= i && i < left;
      @     (\forall int j; left <= j && j < a.length;
      @         a[i] <= a[j]));
      @ requires (\forall int i; 0 <= i && i <= right;
      @     (\forall int j; right < j && j < a.length;
      @         a[i] <= a[j]));
      @ ensures a[e1] <= a[e2] && a[e2] <= a[e3]
      @      && a[e3] <= a[e4] && a[e4] <= a[e5];
      @ ensures (\forall int i; 0 <= i && i < left;
      @     (\forall int j; left <= j && j < a.length;
```

```
      @          a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right;
      @      (\forall int j; right < j && j < a.length;
      @          a[i] <= a[j]));
      @ assignable a[left..right];
      @*/
    {
      /*@
        @ ensures (a[e1] <= a[e2]);
        @ ensures (\forall int i; 0 <= i && i < left;
        @      (\forall int j; left <= j && j < a.length;
        @          a[i] <= a[j]));
        @ ensures (\forall int i; 0 <= i && i <= right;
        @      (\forall int j; right < j && j < a.length;
        @          a[i] <= a[j]));
        @ assignable a[e1], a[e2];
        @ signals_only \nothing;
        @*/
      {
      if (a[e2] < a[e1]) { int t = a[e2]; a[e2] = a[e1]; a[e1
          ] = t; }
      }

      /*@
        @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]);
        @ ensures (\forall int i; 0 <= i && i < left;
        @      (\forall int j; left <= j && j < a.length;
        @          a[i] <= a[j]));
        @ ensures (\forall int i; 0 <= i && i <= right;
        @      (\forall int j; right < j && j < a.length;
        @          a[i] <= a[j]));
        @ assignable a[e1], a[e2], a[e3];
        @ signals_only \nothing;
        @*/
      {
      if (a[e3] < a[e2]) { int t = a[e3]; a[e3] = a[e2]; a[e2
          ] = t;
           if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
      }}

      /*@
        @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]
        @         && a[e3] <= a[e4]);
        @ ensures (\forall int i; 0 <= i && i < left;
        @      (\forall int j; left <= j && j < a.length;
        @          a[i] <= a[j]));
        @ ensures (\forall int i; 0 <= i && i <= right;
```

```
          @     (\forall int j; right < j && j < a.length;
          @         a[i] <= a[j]));
          @ assignable a[e1], a[e2], a[e3], a[e4];
          @ signals_only \nothing;
          @*/
         {
         if (a[e4] < a[e3]) { int t = a[e4]; a[e4] = a[e3]; a[e3
            ] = t;
             if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
                 if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
             }
         }}

         /*@
          @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]
          @       && a[e3] <= a[e4] && a[e4] <= a[e5]);
          @ ensures (\forall int i; 0 <= i && i < left;
          @     (\forall int j; left <= j && j < a.length;
          @         a[i] <= a[j]));
          @ ensures (\forall int i; 0 <= i && i <= right;
          @     (\forall int j; right < j && j < a.length;
          @         a[i] <= a[j]));
          @ assignable a[e1], a[e2], a[e3], a[e4], a[e5];
          @ signals_only \nothing;
          @*/
         {
         if (a[e5] < a[e4]) { int t = a[e5]; a[e5] = a[e4]; a[e4
            ] = t;
             if (t < a[e3]) { a[e4] = a[e3]; a[e3] = t;
                 if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
                     if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t;
                        }
                 }
             }
         }}
       }
    }
}
```

Listing B.2: DualPivotQuicksort_sort_blocks.java

B.1.2. Comparison Between the Block Contract Rules

See 7.1.3.

```
public class DualPivotQuicksort_sort_external {

    static int less, great;
```

```java
static int e1,e2,e3,e4,e5;

/*@
  @ normal_behaviour
  @ requires a.length > 46;
  @ requires 0 <= left && left < e1
  @        && e5 < right && right < a.length;
  @ requires left < e1 && e1 < e2 && e2 < e3
  @        && e3 < e4 && e4 < e5 && e5 < right;
  @ requires (\forall int i; 0 <= i && i < left;
  @      (\forall int j; left <= j && j < a.length;
  @          a[i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right;
  @      (\forall int j; right < j && j < a.length;
  @          a[i] <= a[j]));
  @ ensures a[e1] <= a[e2] && a[e2] <= a[e3]
  @       && a[e3] <= a[e4] && a[e4] <= a[e5];
  @ ensures (\forall int i; 0 <= i && i < left;
  @      (\forall int j; left <= j && j < a.length;
  @          a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @              (\forall int j; right < j && j < a.length;
  @                  a[i] <= a[j]));
  @ assignable a[left..right];
  @*/
static void eInsertionSort(
        int[] a, int left, int right,
        int e1, int e2, int e3, int e4, int e5) {
    /*@ requires a != null;
      @ requires 0 <= left && left < e1
      @        && e5 < right && right < a.length;
      @ requires left < e1 && e1 < e2 && e2 < e3
      @        && e3 < e4 && e4 < e5 && e5 < right;
      @ requires (\forall int i; 0 <= i && i < left;
      @      (\forall int j; left <= j && j < a.length;
      @        a[i] <= a[j]));
      @ requires (\forall int i; 0 <= i && i <= right;
      @      (\forall int j; right < j && j < a.length;
      @       a[i] <= a[j]));
      @ ensures (a[e1] <= a[e2]);
      @ ensures (\forall int i; 0 <= i && i < left;
      @      (\forall int j; left <= j && j < a.length;
      @        a[i] <= a[j]));
      @ ensures (\forall int i; 0 <= i && i <= right;
      @      (\forall int j; right < j && j < a.length;
      @        a[i] <= a[j]));
      @ assignable a[e1], a[e2];
```

```
  @ signals_only \nothing;
  @*/
{
if (a[e2] < a[e1]) { int t = a[e2]; a[e2] = a[e1]; a[e1]
    = t; }
}

/*@ requires a != null;
  @ requires 0 <= left && left < e1
  @         && e5 < right && right < a.length;
  @ requires left < e1 && e1 < e2 && e2 < e3
  @         && e3 < e4 && e4 < e5 && e5 < right;
  @ requires (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @         a[i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @         a[i] <= a[j]));
  @ requires (a[e1] <= a[e2]);
  @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]);
  @ ensures (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @         a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @         a[i] <= a[j]));
  @ assignable a[e1], a[e2], a[e3];
  @ signals_only \nothing;
  @*/
{
if (a[e3] < a[e2]) { int t = a[e3]; a[e3] = a[e2]; a[e2]
    = t;
     if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
}}

/*@ requires a != null;
  @ requires 0 <= left && left < e1
  @         && e5 < right && right < a.length;
  @ requires left < e1 && e1 < e2 && e2 < e3
  @         && e3 < e4 && e4 < e5 && e5 < right;
  @ requires (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @         a[i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @         a[i] <= a[j]));
  @ requires (a[e1] <= a[e2] && a[e2] <= a[e3]);
```

```
  @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]
  @            && a[e3] <= a[e4]);
  @ ensures (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @        a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @        a[i] <= a[j]));
  @ assignable a[e1], a[e2], a[e3], a[e4];
  @ signals_only \nothing;
  @*/
{
if (a[e4] < a[e3]) { int t = a[e4]; a[e4] = a[e3]; a[e3]
    = t;
     if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
          if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
     }
}}

/*@ requires a != null;
  @ requires 0 <= left && left < e1
  @        && e5 < right && right < a.length;
  @ requires left < e1 && e1 < e2 && e2 < e3
  @        && e3 < e4 && e4 < e5 && e5 < right;
  @ requires (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @          a[i] <= a[j]));
  @ requires (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @          a[i] <= a[j]));
  @ requires (a[e1] <= a[e2] && a[e2] <= a[e3]
  @         && a[e3] <= a[e4]);
  @ ensures (a[e1] <= a[e2] && a[e2] <= a[e3]
  @        && a[e3] <= a[e4] && a[e4] <= a[e5]);
  @ ensures (\forall int i; 0 <= i && i < left;
  @     (\forall int j; left <= j && j < a.length;
  @        a[i] <= a[j]));
  @ ensures (\forall int i; 0 <= i && i <= right;
  @     (\forall int j; right < j && j < a.length;
  @        a[i] <= a[j]));
  @ assignable a[e1], a[e2], a[e3], a[e4], a[e5];
  @ signals_only \nothing;
  @*/
{
if (a[e5] < a[e4]) { int t = a[e5]; a[e5] = a[e4]; a[e4]
    = t;
     if (t < a[e3]) { a[e4] = a[e3]; a[e3] = t;
```

```
                    if (t < a[e2]) { a[e3] = a[e2]; a[e2] = t;
                        if (t < a[e1]) { a[e2] = a[e1]; a[e1] = t; }
                    }
                }
            }}
        }
}
```

Listing B.3: DualPivotQuicksort_sort_external

B.2. Loop Contracts

B.2.1. List Increment

See 7.2.2.

```
public interface IntList {

    /*@ public ghost \locset footprint; */
    /*@ public ghost \seq seq; */

    /*@ public invariant \subset(
      @      \singleton(this.seq), footprint);
      @ public invariant \subset(
      @      \singleton(this.footprint), footprint);
      @ public invariant (
      @      \forall int i; 0<=i && i<seq.length;
      @          seq[i] instanceof int);
      @ public accessible \inv: footprint;
      @*/
}
```

Listing B.4: IntList.java

```

public final class IntNode {
    public /*@ nullable @*/ IntNode next;
    public int data;
}
```

Listing B.5: IntNode.java

```
public final class IntLinkedList implements IntList {

    /*@ nullable @*/ IntNode first;
    /*@ nullable @*/ IntNode last;
    int size;

```

```
/*@ ghost \seq nodeseq; */

/*@ invariant footprint == \set_union(this.*,
  @       \infinite_union(int i; 0<=i && i<size;
  @           ((IntNode)nodeseq[i]).*));
  @
  @ invariant (\forall int i; 0<=i && i<size;
  @           ((IntNode)nodeseq[i]) != null
  @       && ((IntNode)nodeseq[i]).data == seq[i]
  @       && (\forall int j; 0<=j && j<size;
  @               (IntNode)nodeseq[i] == (IntNode)nodeseq[j]
  @                   ==> i == j)
  @       && ((IntNode)nodeseq[i]).next == (i==size-1
  @           ? null : (IntNode)nodeseq[i+1]));
  @
  @ invariant first == (size == 0
  @       ? null : (IntNode)nodeseq[0]);
  @ invariant last == (size == 0
  @      ? null : (IntNode)nodeseq[size-1]);
  @
  @ invariant size == seq.length && size == nodeseq.length;
  @*/

/*@ normal_behavior
  @ ensures (\forall int i; 0 <= i && i < size;
  @      ((int) seq[i]) == \old((int) seq[i]) + 1);
  @ ensures size == \old(size);
  @ assignable \set_union(\singleton(seq),
  @      \infinite_union(int j; 0 <= j && j < size;
  @          \singleton(((IntNode)nodeseq[j]).data)));
  @*/
public void mapIncrement_loopContract() {
    IntNode current = first;
    int i = 0;

    /*@ loop_contract normal_behavior
      @ requires \invariant_for(this);
      @ requires 0 <= i && i <= size;
      @ requires i < size
      @      ==> current == (IntNode) nodeseq[i];
      @ requires i == size ==> current == null;
      @ ensures \invariant_for(this);
      @ ensures (\forall int j; \before(i) <= j && j < size;
      @      (int) seq[j] == \before((int) seq[j]) + 1);
      @ ensures size == \before(size);
      @ assignable \set_union(\singleton(seq),
      @      \infinite_union(int j; 0 <= j && j < size;
```

```
      @          \singleton(((IntNode)nodeseq[j]).data)));
      @ decreases nodeseq.length - i;
      @*/
    {
      while (current != null) {
        ++current.data;
        //@ set seq = \seq_concat(\seq_sub(seq, 0, i),
           \seq_concat(\seq_singleton(current.data),
           \seq_sub(seq, i+1, size)));
        current = current.next;
        ++i;
      }
    }
}

/*@ normal_behavior
  @ ensures (\forall int i; 0 <= i && i < size;
  @     ((int) seq[i]) == \old((int) seq[i]) + 1);
  @ ensures size == \old(size);
  @ assignable \set_union(\singleton(seq),
  @     \infinite_union(int j; 0 <= j && j < size;
  @         \singleton(((IntNode)nodeseq[j]).data)));
  @*/
public void mapIncrement_loopInvariant() {
    IntNode current = first;
    int i = 0;

    /*@ loop_invariant \invariant_for(this);
      @ loop_invariant 0 <= i && i <= size;
      @ loop_invariant i < size
      @     ==> current == (IntNode) nodeseq[i];
      @ loop_invariant i == size ==> current == null;
      @ loop_invariant (\forall int j; 0 <= j && j < i;
      @     (int) seq[j] == \old((int) seq[j]) + 1);
      @ loop_invariant (\forall int j; i <= j && j < size;
      @     (int) seq[j] == \old((int) seq[j]));
      @ loop_invariant size == \old(size);
      @ assignable \set_union(\singleton(seq),
      @     \infinite_union(int j; 0 <= j && j < size;
      @         \singleton(((IntNode)nodeseq[j]).data)));
      @ decreases nodeseq.length - i;
      @*/
    while (current != null) {
        ++current.data;
        //@ set seq = \seq_concat(\seq_sub(seq, 0, i),
           \seq_concat(\seq_singleton(current.data), \seq_sub
           (seq, i+1, size)));
```

```
                current = current.next;
                ++i;
            }
        }
}
```

Listing B.6: IntLinkedList.java