Chapter 3
Dynamic Logic for Java

Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß

3.1 Introduction

In the previous chapter, we have introduced JFOL a variant of classical first-order logic tailored for reasoning about (single) states of Java programs (Section 2.4). Now, we extend this logic such that we can reason about the behavior of programs, which requires to consider not just one but several program states. As a trivial example, consider the Java statement `x++`. We want to be able to express that this statement, when started in a state where `x` is zero, terminates in a state where `x` is one.

We use an instance of dynamic logic (DL) [Harel, 1984, Harel et al., 2000, Kozen and Tiuryn, 1990, Pratt, 1977] for this purpose, which we will call JavaDL. The principle of dynamic logic is the formulation of assertions about program behavior by integrating programs and formulas within a single language. To this end, the *modalities* $\langle p \rangle$ and $[p]$ can be used in formulas, where p can be any sequence of legal program statements (i.e., DL is a multi-modal logic). These operators refer to the final state of p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that, *if* p terminates, then ϕ holds in the final state. For example, "when started in a state where `x` is zero, `x++` terminates in a state where `x` is one" can in DL be expressed as $\mathtt{x} \doteq 0 \rightarrow \langle \mathtt{x++} \rangle (\mathtt{x} \doteq 1)$.

Nondeterministic programs can have more than one final state; but here, since we consider Java programs to be deterministic, there is exactly one final state (if p terminates normally, i.e., does not terminate abruptly due to an uncaught exception) or there is no final state (if p does not terminate or terminates abruptly). "Deterministic" here means that a program, for the same initial state and the same inputs, always has the same behavior—in particular, the same final state (if it terminates) and the same outputs. Assuming Java to be deterministic is justified as we do not consider concurrency, which is the main source of nondeterminism in Java.

In exact terms, the programming language supported by JavaDL, as defined in this chapter, is not full Java. It lacks features like concurrency, floating-point arithmetic, and dynamic class loading, but retains the essentials of object-orientation. In fact, JavaDL supports all features that occur in both Java Card (version 2.2.2 or 3.0.x, classic edition)—a Java dialect for smart cards—and Java (version 1.4). Beyond Java Card features, JavaDL supports Java's dynamic object creation and initialization, assertions, the primitive types `char` and `long`, strings, multi-dimensional arrays, the enhanced `for`-loop, and more. Extending JavaDL to cover Java Card-specific extensions like transactions is the topic of Chapter 10.

Deduction in DL, and in particular in JavaDL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of Java (see Section 3.5.6).

Dynamic Logic and Hoare Logic

Dynamic logic can be seen as an extension of Hoare logic. The DL formula $\phi \rightarrow [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. Using a program in ϕ, for example, it is easy to specify that an input data structure is not cyclic, which is impossible in pure first-order logic.

A version of KeY that, for teaching purposes, supports a variant of Hoare logic, is described in Chapter 17.

Structure of this Chapter

We first define syntax and semantics of JavaDL in Sections 3.2 and 3.3, respectively. In Section 3.4, we add another type of modal operators to JavaDL, called *updates*, that (like programs) can be used to describe state changes. Then, in Sections 3.5–3.7, we present the JavaDL calculus, which is used in the KeY system for verifying Java programs. Section 3.5 gives an overview, Section 3.6 describes the basic rules of the calculus, and Section 3.7 gives an introduction to the rules for unbounded loops and replacing method invocations by specifications. These latter rules use program abstraction, which is described in more detail in Chapter 9.

3.2 Syntax of JavaDL

In this section, we define the syntax—and later in the chapter, semantics—of JavaDL for a given Java program *Prg*. By *Java program* we mean, as usual, a set of source files containing a set of class definitions. We assume that *Prg* can be compiled without errors.

It is worth noting that while the syntax and semantics of the logic are tied to a fixed and completely known program, the calculus is “modular” and does not have this restriction. Individual methods are soundly verified without the rest of the program being taken into particular consideration—unless the user deliberately chooses to forego modularity.

3.2.1 Type Hierarchies

The minimal type hierarchy $\mathscr{T}_J$ for JFOL was already introduced in Section 2.4.1. A JavaDL type hierarchy for a given Java program *Prg* is any hierarchy $\mathscr{T} = (\mathrm{TSym}, \sqsubseteq)$ that contains $\mathscr{T}_J$ as a subhierarchy (see Figure 2.3 on page 38). That is, it contains (at least) the class and interface types from *Prg* in addition to the types *Any*, *boolean*, *int*, *Null*, *LocSet*, *Field*, *Heap*, $\perp$, $\top$.

We map the finite-width Java integer types `byte`, `short`, `int`, etc. to the unbounded JavaDL type $\mathit{int} \in \mathrm{TSym}$. This mapping does *not* necessarily mean that integer overflows are ignored. Instead, the handling of overflow depends on the semantics and rules for reasoning about the arithmetical operators of Java, which are configurable in KeY. The KeY system allows the user to choose between several different ways of reasoning about the Java integers: (i) ignoring integer overflows, (ii) checking that no integer overflows can occur, and (iii) using the actual modulo semantics of Java. The details can be found in Section 5.4 and, ultimately, in [Beckert and Schlager, 2004, 2005].

Note that Java Card and KeY do not support the Java floating-point types `float` and `double`, so there are also no corresponding types in $\mathscr{T}_J$.

3.2.2 Signatures

In JavaDL, symbols can be either *rigid* or *nonrigid*. The intuition is that the interpretation of nonrigid symbols can be changed by the program, while rigid symbols maintain their interpretation throughout program execution. The class of nullary nonrigid function symbols has a particular importance—we will refer to such symbols as *program variables*.

Definition 3.1. Let $\mathscr{T}$ be a JavaDL type hierarchy for a Java program *Prg*. A JavaDL *signature* w.r.t. $\mathscr{T}$ is a tuple

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

where

- $(\mathrm{FSym}, \mathrm{PSym}, \mathrm{VSym})$ is a JFOL signature, i.e., Σ includes the vocabulary from Σ_J (see Figure 2.4);

- the set ProgVSym of nullary nonrigid function symbols, which we call *program variables*, contains all local variables `a` declared in *Prg*, where the type of $\mathtt{a}{:}A \in \text{ProgVSym}$ is given by the declared Java type T as follows:
  - $A = T$ if T is a reference type,
  - $A = \mathit{boolean}$ if $T = \mathtt{boolean}$,
  - $A = \mathit{int}$ if $T \in \{\mathtt{byte}, \mathtt{short}, \mathtt{int}, \mathtt{long}, \mathtt{char}\}$.
- ProgVSym contains an infinite number of symbols of every typing.
- ProgVSym contains the “special” program variable

$$\mathtt{heap}{:}\mathit{Heap} \in \text{ProgVSym} \quad .$$

There is an important difference between *logical variables* in VSym and *program variables* in ProgVSym: logical variables can be universally or existentially quantified but never occur in programs, while program variables can occur in programs but cannot be quantified.

3.2.3 Syntax of JavaDL Program Fragments

The programs p occurring in modal operators $\langle p \rangle$ and $[p]$ in JavaDL formulas are written in Java, or, more precisely, in the intersection between Java and Java Card. Thus, for full formal rigor, the definitions of JavaDL would have to include definitions of the syntax and semantics of this subset of Java. However, this is beyond the scope of this text. Instead, Definition 3.2 below defines the admissible programs p rather informally, by referring to the *Java language specification* (JLS) [Gosling et al., 2013].

Definition 3.2 (Legal program fragments). Let *Prg* be a Java program. A *legal program fragment p in the context of Prg* is a sequence of Java statements, where there are local variables $\mathtt{a}_1, \ldots, \mathtt{a}_n \in \text{ProgVSym}$ of Java types $T_1, \ldots, T_n$ such that extending *Prg* with an additional class

```
class C {
    static void m(T1 a1, ..., Tn an) throws Throwable { p }
}
```

yields a legal program according to the rules of the Java language specification (with certain deviations outlined below).

The purpose of the parameter declarations T_1 $\mathtt{a}_1$, ..., T_n $\mathtt{a}_n$ of `m` is to bind *free* occurrences of the program variables $\mathtt{a}_1, \ldots, \mathtt{a}_n$ in p, i.e., occurrences not bound by a declaration within p itself. For example, in the legal program fragment “`int a = b;`” there is a free occurrence of the program variable $\mathtt{b} \in \text{ProgVSym}$. The `throws Throwable` clause is included to accommodate any uncaught checked exceptions originating from p.

The deviations from the program legality in the sense of the JLS, include the following syntactical extensions:

- p may contain *method frames* in addition to normal Java statements. A method frame is a statement of the form

  `method-frame(result=`r`,source=`m`(`T_1`,...,`T_n`)@`T`,this=`t`):{` *body* `}`

  where (a) r is a local variable (in case of a void method `result=`r is omitted), (b) m`(`T_1`,...,`T_n`)@`T is a class and method context (method m with given signature of class T), (c) t is an expression free of side-effects and without method calls, and (d) *body* is a legal program fragment in the context of *Prg*. The semantics of a method frame is that, inside *body* (but outside of any nested method frames that might be contained in *body*), the visibility rules for the given class and method context m`(`T_1`,...,`T_n`)@`T are applicable, keyword `this` evaluates to the value of t, and the meaning of a `return` statement is to assign the returned value to r and to then exit the method frame.
- p may contain *method body statements*

  $$retvar\texttt{=}target\texttt{.}m\texttt{(}t_1\texttt{,...,}t_n\texttt{)@}T\texttt{;}$$

  where

  – $target\texttt{.}m\texttt{(}t_1,\ldots,t_n\texttt{)}$ is a method invocation expression,
  – the type T points to a class declared in *Prg*,
  – the result of the method is assigned to *retvar* after return (if the method is not void).

  Intuitively, a method body statement is a shorthand notation for the precisely identified implementation of method $m(\ldots)$ in class T (in other words, for the unambiguously resolved corresponding method invocation). In contrast to a normal method call where the implementation to be taken is determined by dynamic binding, a method body statement is a call to a method declared in a type that is precisely identified by the method body statement.

Typically, method body statements are already contained in initial proof obligations for functional contracts (Definition 8.4), while method frames are only created during symbolic execution.

We also deviate from the JLS by relaxing its requirements in certain aspects, among them:

- Outside method frames, p may refer to fields, methods, and classes that are not visible in `C`. Inside a method frame, KeY follows the visibility rules of the JLS, except that when resolving a method invocation by inlining, the inlined code may refer to classes not visible in the calling method.
- We do not require *definite assignment*. In Java, the value of a local variable or a `final` field must have a definitely assigned value when any access of its value occurs [Gosling et al., 2013, Section 16]. In JavaDL we allow sequences of

statements that violate this condition (the variable then has a well-defined but unknown value).

- We do not ban *unreachable statements* [Gosling et al., 2013, Section 14.21]. For example, we consider

```
throw new RuntimeException(); int i = 0;
```

a legal program fragment.

3.2.4 Syntax of JavaDL Terms and Formulas

JavaDL *terms* are defined in the same way as FOL terms (Definition 2.3). However, the resulting set of terms is a strict superset of the terms of FOL, as the definitions of terms and formulas are mutually recursive, and JavaDL admits formulas that contain the modal operators $\langle p\rangle$ and $[p]$ and are, thus, not part of FOL.

Definition 3.3 (Terms and Formulas of JavaDL). Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg*, and Σ a signature w.r.t. $\mathscr{T}$.

The set DLTrm_A of JavaDL terms of type A, for $A \neq \perp$, and the set DLFml of JavaDL formulas are defined as in first-order logic (Definitions 2.3 and 2.4, page 24) except for the following differences:

- The signature Σ now refers to the JavaDL signature.
- The mutual recursive references to Trm_X and Fml are now to DLTrm_X and DLFml, respectively.
- The following fourth clause is added to the definition of formulas:

  4. $\langle p\rangle\phi,\ [p]\phi \in \mathrm{DLFml}$ for all legal program fragments p.

A term or formula is called rigid if it does not contain any occurrences of program variables.

We use the shorthand notation $o.a$ for $select_A(\mathtt{heap}, o, a)$, where the declared type of attribute a is A. Similarly, $a[i]$ is shorthand for $select_A(\mathtt{heap}, a, arr(i))$. These notations are also used by the KeY pretty printer; see Section 16.2.

Definition 3.4. The definition of the sets *var* of variables and *fv* of free variables in a term or formula is extended to JavaDL by adding the following clauses to the FOL version of their definition (Definition 2.5):

- $var(\mathtt{a}) = \emptyset, fv(\mathtt{a}) = \emptyset$ for $\mathtt{a} \in \mathrm{ProgVSym}$
- $var(\langle p\rangle\phi) = var(\phi), fv(\langle p\rangle\phi) = fv(\phi)$ for $\phi \in \mathrm{DLFml}$
- $var([p]\phi) = var(\phi), fv([p]\phi) = fv(\phi)$ for $\phi \in \mathrm{DLFml}$

3.3 Semantics

To define the syntax of JavaDL, we have extended first-order logic with program variables and program modalities. On the semantic level, the difference is that JavaDL formulas are not evaluated in a single first-order structure but in a so-called Kripke structure, which is a collection of first-order structures.

3.3.1 Kripke Structures

Different first-order structures within a Kripke structure assign different values to program variables. Accordingly, they are called *program states* or simply *states*. We demand that states in the *same* Kripke structure differ only in the interpretation of the *nonrigid* symbols (i.e., the program variables). Two different Kripke structures, on the other hand, may differ in the choice of domain or interpretation of the predicate and (rigid) function symbols.

Definition 3.5 (JavaDL Kripke structure). Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg* and Σ a signature w.r.t. $\mathscr{T}$. A *JavaDL Kripke structure* for Σ is a tuple

$$\mathscr{K} = (\mathscr{S}, \rho)$$

consisting of

- an infinite set $\mathscr{S}$ of first-order structures over Σ (Definition 2.13), which we will call *states*, such that:
  - Any two states $s_1, s_2 \in \mathscr{S}$ coincide in their domain and in the interpretation of predicate and function symbols.
  - $\mathscr{S}$ is closed under the above property, i.e., any FOL structure coinciding with the states in $\mathscr{S}$ in the domain and the interpretation of the predicate and function symbols is also in $\mathscr{S}$.
- a function ρ that associates with every legal program fragment p a *transition relation* $\rho(p) \subseteq \mathscr{S} \times \mathscr{S}$ such that $(s_1, s_2) \in \rho(p)$ iff p, when started in s_1, terminates normally in s_2 (i.e., not by throwing an exception). (We consider Java programs to be deterministic, so for all legal program fragments p and all $s_1 \in \mathscr{S}$, there is at most one s_2 such that $(s_1, s_2) \in \rho(p)$.)

Here, we do not give a formal definition of the transition relation ρ and, thus, no formalization of the semantics of Java. Instead, we treat the function ρ as a black box that captures the behavior of the legal program fragments p and is informally described by the Java Language Specification [Gosling et al., 2013]. We do, however, explicitly formalize the behavior of Java programs on the level of the calculus, in the form of symbolic execution rules (Section 3.6).

The fact that all states of a JavaDL Kripke structure $\mathscr{K}$ share a common domain is sometimes referred to as the *constant domain assumption*. This simplifies, for example, reasoning about quantifiers in the presence of modal operators and updates. On the other hand, the Java programs appearing in formulas may allocate new objects (i.e., elements of D^{Object}) that did not exist previously. This apparent contradiction is resolved with the help of the special field *created*: given a heap $h \in D^{Heap}$ and an object $o \in D^{Object}$, the object o is considered "created" in h in the sense of Java if and only if *created* is set to true for this object in h, i.e., if $h\big(o, I(created)\big) = tt$. An allocation statement in a program is understood as choosing a previously noncreated object in D^{Object}, and setting its *created* field to true in the heap. The alternative of abandoning the constant domain assumption has been investigated by Ahrendt et al. [2009b].

3.3.2 Semantics of JavaDL Terms and Formulas

Similar to the first-order case, we inductively define the semantics of JavaDL terms and formulas. Since program variables can have different meanings in different states, the valuation function is parameterized with a Kripke structure $\mathscr{K}$ and a state s in $\mathscr{K}$.

The semantics of terms and formulas without modalities matches that of first-order logic.

Definition 3.6 (Semantics of JavaDL terms and formulas). Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg*, Σ a signature w.r.t. $\mathscr{T}$, $\mathscr{K} = (\mathscr{S}, \rho)$ a Kripke structure for Σ, $s \in \mathscr{S}$ a state, and $\beta :$ VSym $\rightarrow D$ a variable assignment.

For every JavaDL term $t \in \mathrm{DLTrm}_A$, we define its evaluation by

$$val_{\mathscr{K},s,\beta}(t) = \mathrm{val}_{s,\beta}(t) \quad ,$$

where $\mathrm{val}_{s,\beta}$ is defined as in the first-order case (Definition 2.15).

For every JavaDL formula $\phi \in \mathrm{Fml}$, we define when ϕ is considered to be true with respect to $\mathscr{K}, s, \beta$, which is denoted with $(\mathscr{K}, s, \beta) \models \phi$, by Clauses 1–9 as shown in the definition of the semantics of FOL formulas (Definition 2.16)—with $\mathscr{M} = s$ and $(\mathscr{K}, s, \beta)$ replaced for $(\mathscr{M}, \beta)$—in combination with the two new clauses:

10 $(\mathscr{K}, s, \beta) \models [\mathtt{p}]\phi$ iff there is no s' with $(s, s') \in \rho(\mathtt{p})$ or $(\mathscr{K}, s', \beta) \models \phi$ for s' with $(s, s') \in \rho(\mathtt{p})$

11 $(\mathscr{K}, s, \beta) \models \langle\mathtt{p}\rangle\phi$ iff there is an s' with $(s, s') \in \rho(\mathtt{p})$ and $(\mathscr{K}, s', \beta) \models \phi$ for s' with $(s, s') \in \rho(\mathtt{p})$

As said above, we consider Java programs to be deterministic, such that there is at most one s' with $(s, s') \in \rho(p)$ for each $s \in \mathscr{S}$.

Finally, we define what it means for a JavaDL formula to be satisfiable, respectively valid. A first-order formula is satisfiable (respectively valid) if it holds in some (all) model(s) for some (all) variable assignment(s). Similarly, a JavaDL formula is satisfiable (respectively valid) if it holds in some (all) state(s) of some (all) Kripke structure(s) $\mathscr{K}$ for some (all) variable assignment(s).

Definition 3.7. Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg*, Σ a signature w.r.t. $\mathscr{T}$, and $\phi \in \mathrm{Fml}$ a formula.

ϕ is *satisfiable* if there is a Kripke structure $\mathscr{K} = (\mathscr{S}, \rho)$, a state $s \in \mathscr{S}$ and a variable assignment β such that $(\mathscr{K}, s, \beta) \models \phi$.

ϕ is *logically valid*, denoted by $\models \phi$, if $(\mathscr{K}, s, \beta) \models \phi$ for all Kripke structures $\mathscr{K} = (\mathscr{S}, \rho)$, all states $s \in \mathscr{S}$, and all variable assignments β.

3.4 Describing Transitions between States: Updates

3.4.1 Syntax and Semantics of JavaDL Updates

JavaDL extends classical logic with another syntactical category besides modal operators with program fragments, namely *updates*. Like program fragments, updates denote state changes. The difference between updates and program fragments is that updates are a simpler and more restricted concept. For example, updates always terminate, and the expressions occurring in updates never have side effects.

Definition 3.8 (Updates). Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg*, and Σ a signature for $\mathscr{T}$. The set Upd of updates is inductively defined by:

- $(\mathtt{a} := t) \in \mathrm{Upd}$ for each program variable symbol $\mathtt{a} : A \in \mathrm{ProgVSym}$ and each term $t \in \mathrm{DLTrm}_{A'}$ such that $A' \sqsubseteq A$.
- $\mathtt{skip} \in \mathrm{Upd}$.
- $(u_1 \,||\, u_2) \in \mathrm{Upd}$ for all updates $u_1, u_2 \in \mathrm{Upd}$.
- $(\{u_1\}\, u_2) \in \mathrm{Upd}$ for all updates $u_1, u_2 \in \mathrm{Upd}$.

An expression of the form $\{u\}$, where $u \in \mathrm{Upd}$, is called an *update application*.

Intuitively, an *elementary update* $\mathtt{a} := t$ assigns the value of the term t to the program variable a. The *empty update* that does not change anything is denoted by skip. A *parallel update* $u_1 \,||\, u_2$ executes the subupdates u_1 and u_2 in parallel (as parallel composition is associative, e.g., $(u_1 \,\|\, (u_2 \,\|\, u_3))$ can be written as $u_1 \,\|\, u_2 \,\|\, u_3$). The semantics of $\{u\}\, x$, i.e., prefixing an expression x with an update application, is that x is to be evaluated in the state produced by the update u (the expression x can be a term, a formula, or another update). The precise definition of the semantics of updates is given in Definition 3.11 below.

We extend the definition of occurring and free variables to include updates, which is straightforward.

Definition 3.9. In extension of Definitions 2.5 and 3.4:

$$\begin{aligned}
&var(\mathtt{a} := t) &&= var(t) &\qquad& fv(\mathtt{a} := t) &&= fv(t)\\
&var(\mathtt{skip}) &&= \emptyset && fv(\mathtt{skip}) &&= \emptyset\\
&var(u_1 \,||\, u_2) &&= var(u_1) \cup var(u_2) && fv(u_1 \,||\, u_2) &&= fv(u_1) \cup fv(u_2)\\
&var(\{u\}\, x) &&= var(u) \cup var(x) && fv(\{u\}\, x) &&= fv(u) \cup fv(x)
\end{aligned}$$

for $\mathtt{a} \in \mathrm{ProgVSym}$, $t \in \mathrm{DLTrm}_\top$ $u, u_1, u_2 \in \mathrm{Upd}$, $x \in \mathrm{DLTrm}_\top \cup \mathrm{DLFml} \cup \mathrm{Upd}$.

To include updates, we extend the definitions of terms and formulas of JavaDL (Definition 3.3) with additional clauses:

Definition 3.10 (Terms and formulas of JavaDL with updates). The definition of terms (Definition 3.3 and Definition 2.3) is extended with a fourth clause:

4. $\{u\}\, t \in \mathrm{DLTrm}_A$ for all updates $u \in \mathrm{Upd}$ and all terms $t \in \mathrm{DLTrm}_A$.

The definition of formulas (Definition 3.3) is extended with a fifth clause:

5. $\{u\}\, \phi \in \mathrm{DLFml}$ for all formulas $\phi \in \mathrm{DLFml}$ and updates $u \in \mathrm{Upd}$.

Updates transform one state into another. The meaning of $\{u\}t$, where u is an update and t is a term, a formula, or an update, is that t is evaluated in the state produced by u. Note the *last-win semantics* of parallel updates $u_1 \,||\, u_2$: if there is a "clash," where u_1 and u_2 attempt to assign conflicting values to a program variable, then the value written by u_2 prevails.

Definition 3.11 (Semantics of JavaDL updates). Let *Prg* be a Java program, $\mathscr{T}$ a type hierarchy for *Prg*, Σ a signature for $\mathscr{T}$, $\mathscr{K}$ a Kripke structure for Σ, $s \in \mathscr{S}$ a state, and $\beta : \mathrm{VSym} \to D$ a variable assignment.

The valuation function $val_{\mathscr{K},s,\beta} : \mathrm{Upd} \to (\mathscr{S} \to \mathscr{S})$ is defined as follows:

$$val_{\mathscr{K},s,\beta}(a := t)(s')(\mathtt{b}) = \begin{cases} val_{\mathscr{K},s,\beta}(t) & \text{if } \mathtt{b} = \mathtt{a} \\ s'(\mathtt{b}) & \text{otherwise} \end{cases}$$

$$\text{for all } s' \in \mathscr{S},\ \mathtt{b} \in \mathrm{ProgVSym}$$

$$val_{\mathscr{K},s,\beta}(\mathtt{skip})(s') = s' \quad \text{for all } s' \in \mathscr{S}$$

$$val_{\mathscr{K},s,\beta}(u_1 \,||\, u_2)(s') = val_{\mathscr{K},s,\beta}(u_2)(val_{\mathscr{K},s,\beta}(u_1)(s')) \quad \text{for all } s' \in \mathscr{S}$$

$$val_{\mathscr{K},s,\beta}(\{u_1\}\, u_2) = val_{\mathscr{K},s',\beta}(u_2) \quad \text{where } s' = val_{\mathscr{K},s,\beta}(u_1)(s)$$

Moreover, the definition of the semantics of JavaDL terms and formulas (Definition 3.6) is extended for terms with the clause

$$val_{\mathscr{K},s,\beta}(\{u\}\, t) = val_{\mathscr{K},s',\beta}(t) \quad \text{where } s' = val_{\mathscr{K},s,\beta}(u)(s)$$

and it is extended for formulas with the clause

$$(\mathscr{K}, s, \beta) \models val_{\mathscr{K},s,\beta}(\{u\}\, \phi) \quad \text{iff} \quad (\mathscr{K}, s', \beta) \models \phi \quad \text{where } s' = val_{\mathscr{K},s,\beta}(u)(s)$$

Table 3.1 Simplification rules for updates

$\{\ldots \,\|\, \mathtt{a} := t_1 \,\|\, \ldots \,\|\, \mathtt{a} := t_2 \,\|\, \ldots\}\; t \leadsto \{\ldots \,\|\, \mathtt{skip} \,\|\, \ldots \,\|\, \mathtt{a} := t_2 \,\|\, \ldots\}\; t$ where $t \in \mathrm{DLTrm}_A \cup \mathrm{DLFml} \cup \mathrm{Upd}$	$\mathsf{dropUpdate}_1$
$\{\ldots \,\|\, \mathtt{a} := t' \,\|\, \ldots\}\; t \leadsto \{\ldots \,\|\, \mathtt{skip} \,\|\, \ldots\}\; t$ where $t \in \mathrm{DLTrm}_A \cup \mathrm{DLFml} \cup \mathrm{Upd}$, $\mathtt{a} \notin fpv(t)$	$\mathsf{dropUpdate}_2$
$\{u\}\; \{u'\}\; t \leadsto \{u \,\|\, \{u\}\; u'\}\; t$ where $t \in \mathrm{DLTrm}_A \cup \mathrm{DLFml} \cup \mathrm{Upd}$	$\mathsf{seqToPar}$
$\{u \,\|\, \mathtt{skip}\}\; t \leadsto \{u\}\; t$ where $t \in \mathrm{DLTrm}_A \cup \mathrm{DLFml} \cup \mathrm{Upd}$	$\mathsf{parallelWithSkip}_1$
$\{\mathtt{skip} \,\|\, u\}\; t \leadsto \{u\}\; t$ where $t \in \mathrm{DLTrm}_A \cup \mathrm{DLFml} \cup \mathrm{Upd}$	$\mathsf{parallelWithSkip}_2$
$\{\mathtt{skip}\}\; t \leadsto t$ where $t \in \mathrm{DLTrm}_A \cup \mathrm{DLFml} \cup \mathrm{Upd}$	$\mathsf{applySkip}$
$\{u\}\; x \leadsto x$ where $x \in \mathrm{VSym} \cup \{true, false\}$	$\mathsf{applyOnRigid}_1$
$\{u\}\; f(t_1, \ldots, t_n) \leadsto f(\{u\}t_1, \ldots, \{u\}t_n)$ where $f \in \mathrm{FSym} \cup \mathrm{PSym}$	$\mathsf{applyOnRigid}_2$
$\{u\}\; (\text{if } \phi \text{ then } t_1 \text{ else } t_2) \leadsto \text{if } \{u\}\; \phi \text{ then } \{u\}\; t_1 \text{ else } \{u\}\; t_2$	$\mathsf{applyOnRigid}_3$
$\{u\}\; \neg\phi \leadsto \neg\{u\}\; \phi$	$\mathsf{applyOnRigid}_4$
$\{u\}\; (\phi_1 \bullet \phi_2) \leadsto \{u\}\; \phi_1 \bullet \{u\}\; \phi_2$ where $\bullet \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$	$\mathsf{applyOnRigid}_5$
$\{u\}\; \mathcal{Q} A\, x; \phi \leadsto \mathcal{Q} A\, x; \{u\}\; \phi$ where $\mathcal{Q} \in \{\forall, \exists\}$, $x \notin fv(u)$	$\mathsf{applyOnRigid}_6$
$\{u\}\; (\mathtt{a} := t) \leadsto \mathtt{a} := \{u\}\; t$	$\mathsf{applyOnRigid}_7$
$\{u\}\; (u_1 \,\|\, u_2) \leadsto (\{u\}\; u_1) \,\|\, (\{u\}\; u_2)$	$\mathsf{applyOnRigid}_8$
$\{\mathtt{a} := t\}\; \mathtt{a} \leadsto t$	$\mathsf{applyOnTarget}$

3.4.2 Update Simplification Rules

The part of the JavaDL calculus that deals with simplification of updates is shown in Table 3.1.

The $\mathsf{dropUpdate}_1$ rule simplifies away an ineffective elementary subupdate of a larger parallel update: if there is an update to the same program variable a further to the right of the parallel composition, then this second elementary update overrides the first due to the last-win semantics of parallel updates (Definition 3.11).

The $\mathsf{dropUpdate}_2$ rule allows dropping an elementary update $\mathtt{a} := t'$ where the term, formula, or update in scope of the update cannot depend on the value of the program variable a, because it does not contain any free occurrences of a. A *free* occurrence of a program variable is any occurrence, except for an occurrence inside a program fragment p that is bound by a declaration within p. In addition to explicit occurrences, we consider program fragments p to always contain an implicit free occurrence of the program variable `heap`. The function $fpv : \mathrm{DLTrm}_A \cup \mathrm{DLFml} \cup \mathrm{Upd} \rightarrow 2^{\mathrm{ProgVSym}}$ is defined accordingly. For example, we have $fpv([\mathtt{int\ a\ =\ b;}](\mathtt{b} \doteq \mathtt{c})) = \{\mathtt{b}, \mathtt{c}, \mathtt{heap}\}$. Java's rules for *definite assignment*

[Gosling et al., 2013, Chapter 16] ensure that within a program fragment p, a declared program variable (such as `a` in the example) is always written before being read, and that the behavior of p thus cannot depend on its initial value.

The seqToPar rule converts a cascade of two update applications—which corresponds to sequential execution of the two updates—into the application of a single parallel update. Due to the last-win semantics for parallel updates, this is possible by applying the first update to the second, and replacing the sequential composition by parallel composition.

The rules $\mathsf{parallelWithSkip}_i$ and applySkip remove the effect-less `skip` update from parallel updates or apply it as identity to any term, formula, or update.

The remaining rules are responsible for applying updates to terms, formulas and (other) updates as substitutions. The various applyOnRigid rules propagate an update to the subterms below a (rigid) operator. Ultimately, the update can either be simplified away with $\mathsf{dropUpdate}_2$, or it remains as an elementary update $\mathtt{a} := t$ applied to the target program variable `a` itself. In the latter case, the term t is substituted for `a` by the applyOnTarget rule.

The only case not covered by the rules in Table 3.1 is that of applying an update to a modal operator, as in $\{u\}\ [p]\phi$ or $\{u\}\ \langle p\rangle\phi$. For these formulas, the program p must first be eliminated using the symbolic execution rules. Only afterwards can the resulting update be applied to ϕ.

3.5 The Calculus for JavaDL

The calculus for JavaDL follows the same basic logical principles as the calculus for first-order logic (FOL) introduced in Chapter 2. We do thus not repeat them here but only explain extensions and restrictions in comparison to the FOL case. The remaining bulk of this chapter is concerned with explaining in detail how the JavaDL calculus formalizes symbolic execution of Java programs.

3.5.1 JavaDL Rule Schemata and First-Order Rules

Since first-order logic (FOL) is part of JavaDL, all the axioms and rule schemata of the first-order calculus introduced in Chapter 2 are also part of the JavaDL and its calculus. This inclusion pertains, inter alia, Figure 2.1 (classical first-order rules), Figure 2.2 (equality rules), Figure 2.5 (integer axioms and rules), and Figure 2.8 (axioms about types). As a consequence, these rules can be applied to JavaDL sequents—even if the formulas to which they are applied are not purely first-order.

Compared to Section 2.2.2 on FOL calculus, we do simplify and generalize the rule schema notation in two ways, though. First, we leave out the explicit context (in form of formula sets Γ and Δ), which is added on-the-fly during rule application.

Second, we extend the notion of context in that, when writing a rule schema, an update that is common to all premisses can be left out as well.

Definition 3.12. If

$$\frac{\begin{array}{c}\phi_1^1,\dots,\phi_{m_1}^1 \Longrightarrow \psi_1^1,\dots,\psi_{n_1}^1\\ \vdots\\ \phi_1^k,\dots,\phi_{m_k}^k \Longrightarrow \psi_1^k,\dots,\psi_{n_k}^k\end{array}}{\phi_1,\dots,\phi_m \Longrightarrow \psi_1,\dots,\psi_n}$$

is an instance of a rule schema, then

$$\frac{\begin{array}{c}\Gamma,\,\mathcal{U}\phi_1^1,\dots,\mathcal{U}\phi_{m_1}^1 \Longrightarrow \mathcal{U}\psi_1^1,\dots,\mathcal{U}\psi_{n_1}^1,\,\Delta\\ \vdots\\ \Gamma,\,\mathcal{U}\phi_1^k,\dots,\mathcal{U}\phi_{m_k}^k \Longrightarrow \mathcal{U}\psi_1^k,\dots,\mathcal{U}\psi_{n_k}^k,\,\Delta\end{array}}{\Gamma,\,\mathcal{U}\phi_1,\dots,\mathcal{U}\phi_m \Longrightarrow \mathcal{U}\psi_1,\dots,\mathcal{U}\psi_n,\,\Delta}$$

is an inference rule of our DL calculus, where $\mathcal{U}$ is the application of an arbitrary syntactic update (it may be empty), and Γ,Δ are finite sets of context formulas.

If, however, the symbol $(*)$ is added to the rule schema, the context $\Gamma,\Delta,\mathcal{U}$ must be empty, i.e., only instances of the schema itself are inference rules. Later in the book we will present a few rules, e.g., the loop invariant rule (Section 3.7.2), where the context cannot be omitted.

Example 3.13. Consider, for example, the rule impRight, which made a first appearance in Figure 2.1 on page 28. In the just introduced notation, the rule schema for this rule takes the following form:

$$\mathsf{impRight}\ \frac{\phi \Longrightarrow \psi}{\Longrightarrow \phi \rightarrow \psi}$$

When this schema is instantiated for JavaDL, a context consisting of Γ,Δ and an update $\mathcal{U}$ can be added, and the schema variables ϕ,ψ can be instantiated with formulas that are not purely first-order. For example, the following is an instance of impRight:

$$\frac{x \doteq 1,\ \{x := 0\}(x \doteq y) \Longrightarrow \{x := 0\}\langle \mathtt{m();}\rangle(y \doteq 0)}{x \doteq 1 \Longrightarrow \{x := 0\}(x \doteq y \rightarrow \langle \mathtt{m();}\rangle(y \doteq 0))}$$

where $\Gamma = (x \doteq 1)$, Δ is empty, and the context update is $\mathcal{U} = \{x := 0\}$.

Due to the presence of modalities and program variables, which do not exist in purely first-order formulas, different parts of a formula may have to be evaluated in different states. Therefore, the application of some first-order rules that rely on the identity of terms in different parts of a formula need to be restricted. That affects rules for universal quantification and equality rules.

3.5.1.1 Restriction of Rules for Universal Quantification

The rules for universal quantification have the following form:

$$\mathsf{allLeft}\ \frac{\forall x.\phi,\ [x/t](\phi) \Longrightarrow}{\forall x.\phi \Longrightarrow} \qquad \mathsf{exRight}\ \frac{\Longrightarrow \exists x.\phi,\ [x/t](\phi)}{\Longrightarrow \exists x.\phi}$$

where $t \in \mathrm{DLTrm}_{A'}$ is a rigid ground term
whose type A' is a subtype of the type A of x

In the first-order case, the term t that is instantiated for the quantified variable x can be an arbitrary ground term. In JavaDL, however, we have to add the restriction that t is a *rigid* ground term (Definition 3.3). The reason is that, though an arbitrary value can be instantiated for x as it is universally quantified, all occurrences of x must have the same value in each individual instantiation.

Example 3.14. The formula $\forall x.(x \doteq 0 \rightarrow \langle \texttt{i++;} \rangle (x \doteq 0))$ is logically valid, but instantiating the variable x with the nonrigid program variable `i` is wrong as it leads to the unsatisfiable formula $\texttt{i} \doteq 0 \rightarrow \langle \texttt{i++;} \rangle (\texttt{i} \doteq 0)$.

In practice, it is often very useful to instantiate a universally quantified variable x with the value of a nonrigid term t. That, however, is not easily possible as a quantified variable, which is a rigid term, must not be instantiated with a nonrigid term. To solve that problem, one can add the logically valid formula $\exists y.(y \doteq t)$ to the left of the sequent, Skolemize that formula, which yields $c_{sk} \doteq t$, and then instantiate x with the rigid constant c_{sk}.

Rules for existential quantification do not have to be restricted because they introduce *rigid* Skolem constants anyway.

3.5.1.2 Restriction of Rules for Equalities

The equality rules (Figure 2.2) are part of the JavaDL calculus but an equality $t_1 \doteq t_2$ may only be used for rewriting if

- both t_1 and t_2 are rigid terms (Definition 3.3), or
- the equality $t_1 \doteq t_2$ and the occurrence of t_i that is being replaced are (a) not in the scope of two different program modalities and (b-1) not in the scope of two different updates or (b-2) in the scope of syntactically identical updates (in fact, it is also sufficient if the two updates are only semantically identical, i.e., have the same effect). This same-update-level property is explained in more detail in Section 4.3.1.

Example 3.15. The sequent

$$\texttt{x} \doteq \texttt{v} + 1 \Longrightarrow \{\texttt{v} := 2\}(\texttt{x} \doteq 3)$$

is satisfiable, but not valid. According to the above restriction on the equality rule, the equality $\texttt{x} \doteq \texttt{v} + 1$ must not be applied to the occurrence of `x` on the right side of the sequent: (a) The terms are nonrigid; and (b) while the equation is not in the scope of any update, the occurrence of x is below an update.

The example demonstrates that this restriction is crucial for soundness of the calculus as, if we allow the equality to be applied, this would lead to the *valid* sequent

$$\mathtt{x} \doteq \mathtt{v}+1 \Longrightarrow \{\mathtt{v} := 2\}(\mathtt{v}+1 \doteq 3) \ .$$

Thus, we would have turned an invalid into a valid sequent.

In the sequent

$$\{\mathtt{v} := 2\}(\mathtt{x} \doteq \mathtt{v}+1) \Longrightarrow \{\mathtt{v} := 2\}(\mathtt{x} \doteq 3) \ ,$$

however, both the equality and the term being replaced occur in the scope of identical updates and, thus, the equality rule can be applied.

3.5.2 Nonprogram Rules for Modalities

The JavaDL calculus contains some rules that apply to modal operators and, thus, are not first-order rules but that are neither related to a particular Java construct.

The most important representatives of this rule class are the following two rules for handling empty modalities:

$$\mathsf{emptyDiamond}\ \frac{\Longrightarrow \phi}{\Longrightarrow \langle\rangle\phi} \qquad \mathsf{emptyBox}\ \frac{\Longrightarrow \phi}{\Longrightarrow [\,]\phi}$$

The rule

$$\mathsf{diamondToBox}\ \frac{\Longrightarrow [p]\phi \qquad \Longrightarrow \langle p\rangle\mathrm{true}}{\Longrightarrow \langle p\rangle\phi}$$

relates the diamond modality to the box modality. It allows one to split a total correctness proof into a partial correctness proof and a separate proof for termination. Note, that this rule is only sound for deterministic programming languages like Java.

3.5.3 Soundness and Completeness of the Calculus

3.5.3.1 Soundness

The most important property of the JavaDL calculus is soundness, i.e., only valid formulas are derivable.

Proposition 3.16 (Soundness). *If a sequent $\Gamma \Longrightarrow \Delta$ is derivable in the JavaDL calculus (Definition 2.10), then it is valid, i.e., the formula $\bigwedge\Gamma \rightarrow \bigvee\Delta$ is logically valid (Definition 3.7).*

It is easy to show that the whole calculus is sound if and only if all its rules are sound. That is, if the premisses of any rule application are valid sequents, then the conclusion is valid as well.

Given the soundness of the existing core rules of the JavaDL calculus, the user can add new rules, whose soundness must then be proven w.r.t. the existing rules (see Section 4.4).

Validating the Soundness of the JavaDL Calculus

So far, we have no intention of formally proving the soundness of the JavaDL calculus, i.e., the core rules that are not user-defined (the soundness of user-defined rules can be verified within the KeY system, see Section 4.4). Doing so would first require a formal specification of the Java language. No *official* formal semantics of Java is available though. Furthermore, proving soundness of the calculus requires the use of a higher-order theorem proving tool, and it is a tedious task due to the high number of rules. Resources saved on a formal soundness proof were instead spent on further improvement of the KeY system. We refer to [Beckert and Klebanov, 2006] for a discussion of this policy and further arguments in its favor. On the other hand, the KeY project performs cross-verification against other Java formalizations to ensure the faithfulness of the calculus.

One such effort compares the KeY calculus with the Bali semantics [von Oheimb, 2001], which is a Java Hoare logic formalized in Isabelle/HOL. KeY rules are translated manually into Bali rules. These are then shown sound with respect to the rules of the standard Bali calculus. The published result [Trentelman, 2005] describes in detail the examination of the rules for local variable assignment, field assignment, and array assignment.

Another validation was carried out by Ahrendt et al. [2005]. A reference Java semantics from [Farzan et al., 2004] was used, which is formalized in Rewriting Logic [Meseguer and Rosu, 2004] and mechanized in the input language of the MAUDE system. This semantics is an executable specification, which together with MAUDE provides a Java interpreter. Considering the nature of this semantics, we concentrated on using it to verify our program transformation rules. These are rules that decompose complex expressions, take care of the evaluation order, etc. (about 45% of the KeY calculus). For the cross-verification, the MAUDE semantics was "lifted" in order to cope with schematic programs like the ones appearing in calculus rules. The rewriting theory was further extended with means to generate valid initial states for the involved program fragments, and to check the final states for equivalence. The result is used in automated validation runs, which is beneficial, since the calculus is constantly extended with new features.

Furthermore, the KeY calculus has been tested against the compiler test suite Jacks (part of the Java compiler Jikes). The suite is a collection of intricate programs covering many difficult features of the Java language. These programs are symbolically executed with the KeY calculus and the output is compared to the reference provided by the suite. To what extent testing of verification systems is able to provide evidence for the correctness of the rule base has been examined in [Beckert et al., 2013].

3.5.3.2 Relative Completeness

Ideally, one would like a program verification calculus to be able to prove all statements about programs that are true, which means that all valid sequents should be derivable. That, however, is *impossible* because JavaDL includes first-order arithmetic, which is already inherently incomplete as established by Gödel's Incompleteness Theorem [Gödel, 1931] (see the box on page 40). Another, equivalent, argument is that a complete calculus for JavaDL would yield a decision procedure for the Halting Problem, which is well-known to be undecidable. Thus, a logic like JavaDL cannot ever have a calculus that is both sound and complete.

Still, it is possible to define a notion of *relative completeness* [Cook, 1978], which intuitively states that the calculus is complete "up to" the inherent incompleteness in its first-order part. A relatively complete calculus contains all the rules that are necessary to prove valid program properties. It only may fail to prove such valid formulas whose proof would require the derivation of a nonprovable first-order property (being purely first-order, its provability would be independent of the program part of the calculus).

Proposition 3.17 (Relative Completeness). *If a sequent $\Gamma \Longrightarrow \Delta$ is valid, i.e., the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is logically valid (Definition 3.7), then there is a finite set Γ_{FOL} of logically valid first-order formulas such that the sequent*

$$\Gamma_{FOL}, \Gamma \Longrightarrow \Delta$$

is derivable in the JavaDL calculus.

The standard technique for proving that a program verification calculus is relatively complete [Harel, 1979] hinges on a central lemma expressing that for all JavaDL formulas there is an equivalent purely first-order formula.

A completeness proof for the object-oriented dynamic logic ODL [Beckert and Platzer, 2006], which captures the essence of JavaDL, is given by Platzer [2004]. ODL captures the essence of JavaDL, consolidating its foundational principles into a concise logic. The ODL programming language is a While language extended with an object type system, object creation, and nonrigid symbols that can be used to represent program variables and object attributes. However, it does not include the many other language features, built-in operators, etc. of Java.

3.5.4 Schema Variables for Program Constructs

The schema variables used in rule schemata are all assigned a kind that determines which class of concrete syntactic elements they represent. In the following sections, we often do not explicitly mention the kinds of schema variables but use the name of the variables to indicate their kind. Table 3.2 gives the correspondence between names of schema variables that represent pieces of Java code and their kinds. In addition, we use the schema variables ϕ, ψ to represent formulas and Γ, Δ to represent sets of formulas. Schema variables of corresponding kinds occur also in the *taclets* used to implement rules in the KeY system (see Section 4.2).

Table 3.2 Correspondence between names of schema variables and their kinds

π	nonactive prefix of Java code (Section 3.5.5)
ω	"rest" of Java code after the active statement (Section 3.5.5)
p, q	Java code (arbitrary sequence of statements)
e	arbitrary Java expression
se	simple expression, i.e., any expression whose evaluation, a priori, does not have any side-effects. It is defined as one of the following: (a) a local variable (b) `this.`a, i.e., an access to an instance attribute via the target expression `this` (or, equivalently, no target expression) (c) an access to a static attribute of the form t`.`a, where the target expression t is a type name or a simple expression (d) a literal (e) a compile-time constant (f) an `instanceof` expression with a simple expression as the first argument (g) a `this` reference (h) expressions of types *LocSet* (location sets), *Seq* (finite sequences) etc., provided that their subexpressions are simple expressions (e.g., $union(r,s)$ is a simple expression if r,s are simple). An access to an instance attribute o`.`a is not simple because a `NullPointerException` may be thrown
nse	nonsimple expression, i.e., any expression that is not simple (see above)
lhs	simple expression that can appear on the left-hand-side of an assignment. This amounts to the items (a)–(c) from above
$v, v_0, \ldots$	local program variables
a	attribute
l	label
$args$	argument tuple, i.e., a tuple of expressions
cs	sequence of catch clauses
$mname$	name of a method
T	type expression
C	name of a class or interface

If a schema variable T representing a type expression is indexed with the name of another schema variable, say e, then it only matches the Java type of the expression with which e is instantiated. For example, "T_w v `=` w" matches the Java code "`int i = j`" if and only if the type of `j` is `int` (and not, e.g., `byte`).

3.5.5 The Active Statement in a Modality

The rules of our calculus operate on the first *active* statement p in a modality $\langle \pi p \omega \rangle$ or $[\pi p \omega]$. The nonactive prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`method-frame(...){`" of method invocation blocks. The prefix is needed to (i) keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately; and (ii) to correctly resolve field and method bindings.

The postfix ω denotes the "rest" of the program, i.e., everything except the nonactive prefix and the part of the program the rule operates on (in particular, ω contains closing braces corresponding to the opening braces in π). For example, if a rule is applied to the following Java block operating on its first active command `i=0;`, then the nonactive prefix π and the "rest" ω are the indicated parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

No Rule for Sequential Composition

In versions of dynamic logic for simple programming languages, where no prefixes are needed, any formula of the form $\langle pq \rangle \phi$ can be replaced by $\langle p \rangle \langle q \rangle \phi$. In our calculus, decomposing of $\langle \pi pq \omega \rangle \phi$ into $\langle \pi p \rangle \langle q \omega \rangle \phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p \omega \rangle \langle \pi q \omega \rangle \phi$ cannot be used either because its semantics is in general different from that of $\langle \pi pq \omega \rangle \phi$.

3.5.6 The Essence of Symbolic Execution

Our calculus works by reducing the question of a formula's validity to the question of the validity of several simpler formulas. Since JavaDL formulas contain programs, the JavaDL calculus has rules that reduce the meaning of programs to the meaning of simpler programs. For this reduction we employ the technique of *symbolic execution* [King, 1976]. Symbolic execution in JavaDL resembles playing an accordion: you make the program longer (though simpler) before you can make it shorter.

For example, to find out whether the sequent[1]

$$\Longrightarrow \langle \texttt{o.next.prev=o;} \rangle \texttt{o.next.prev} \doteq \texttt{o}$$

[1] The expression `o.next.prev` is shorthand for $select_A(\texttt{heap}, select_A(\texttt{heap}, o, \texttt{next}), \texttt{prev})$; see Section 3.2.4 and 16.2.

is valid, we symbolically execute the Java code in the diamond modality. At first, the calculus rules transform it into an equivalent but longer—albeit in a sense simpler—sequence of statements:

$$\Longrightarrow \langle\texttt{ListEl v; v=o.next; v.prev=o;}\rangle\texttt{o.next.prev} \doteq \texttt{o} \ .$$

This way, we have reduced the reasoning about the expression `o.next.prev=o` to reasoning about several simpler expressions. We call this process *unfolding*, and it works by introducing fresh local variables to store intermediate computation results.

Now, when analyzing the first of the simpler assignments (after removing the variable declaration), one has to consider the possibility that evaluating the expression `o.next` may produce a side effect if `o` is `null` (in that case an exception is thrown). However, it is not possible to unfold `o.next` any further. Something else has to be done, namely a case distinction. This results in the following two new goals:

$$\texttt{o} \not\doteq \texttt{null} \Longrightarrow \{\texttt{v} := \texttt{o.next}\}\langle\texttt{v.prev=o;}\rangle\texttt{o.next.prev} \doteq \texttt{o}$$
$$\texttt{o} \doteq \texttt{null} \Longrightarrow \langle\texttt{throw new NullPointerException();}\rangle\texttt{o.next.prev} \doteq \texttt{o}$$

Thus, we can state the essence of symbolic execution: the Java code in the formulas is step-wise unfolded and replaced by case distinctions and syntactic updates.

Of course, it is not a coincidence that these two ingredients (case distinctions and updates) correspond to two of the three basic programming constructs. The third basic construct are loops. These cannot in general be treated by symbolic execution, since using symbolic values (as opposed to concrete values), the number of loop iterations is unbounded. Symbolically executing a loop, which is called "unwinding," is useful and even necessary, but unwinding cannot eliminate a loop in the general case. To treat arbitrary loops, one needs to use induction or loop invariants (see Section 3.7.2). (A different method for treating certain loops of a simple, uniform structure is described in [Gedell and Hähnle, 2006].)

Method invocations can be symbolically executed, replacing a method call by the method's implementation. However, it is often useful to instead use a method's contract so that it is only symbolically executed once—during the proof that the method satisfies its contract—instead of executing it for each invocation.

3.5.7 Components of the Calculus

Our JavaDL calculus has several major components, which are described throughout this book. However, since the calculus, as implemented in the KeY system, consists of hundreds of rules, we cannot list them all in this book. Instead, we give typical examples for the different rule types and classes.

The major components of the JavaDL calculus are:

1. Nonprogram rules, i.e., rules that are not related to particular program constructs. This component contains first-order rules (see Chapter 2), which include rules

for reasoning about heaps; rules for data-types, such as integers, sequences and strings (see Chapter 5); rules for modalities (e.g., rules for empty modalities); and the induction rule.

2. Update simplification rules (see Section 3.4.2).
3. Rules for symbolic execution of programs. These rules work towards reducing/simplifying the program and replacing it by a combination of case distinctions (proof branches) and sequences of updates. These rules always (and only) apply to the first active statement. Note that a "simpler" program *may* be syntactically longer; it is simpler in the sense that expressions are not as deeply nested or have less side-effects.

   When presenting these rules, we usually only give the rule versions for the diamond modality $\langle\cdot\rangle$. The rules for box modality $[\cdot]$ are mostly the same—notable exceptions are the rules for handling abrupt termination (Section 3.6.7) and the loop invariant rule that, in fact, belongs to the next component.
4. Rules for program abstraction and modularization. This component contains the loop invariant rule for reasoning about loops for which no fixed upper bound on the number of iterations exists and the rules that replace a method invocation by the method's contract (Section 3.7, see also Chapter 9).

Component 3 is the core for handling Java programs occurring in formulas. These rules can be applied automatically, and they can do everything needed for handling programs except evaluating loops and using method specifications.

The overall strategy for proving a formula containing a program is to use the rules in Component 3, interspersed with applications of rules in Component 4 for handling loops and methods, to step-wise eliminate the program and replace it by updates and case distinctions. After each step, Component 2 is used to simplify/eliminate updates. The final result of this process are sequents containing pure first-order formulas. These are then handled by Component 1.

The symbolic execution process is, for the most part, done automatically by the KeY system. Usually, only handling loops and methods may require user interaction. Also, for solving the first-order problems that are left at the end of the symbolic execution process, the KeY system often needs support from the user (or from the decision procedures integrated into KeY, see Chapter 15).

3.6 Rules for Symbolic Execution of Java Programs

3.6.1 The Basic Assignment Rule

In Java—like in other object-oriented programming languages—different object variables can refer to the same object. This phenomenon, called aliasing, causes serious difficulties for handling assignments in a calculus (a similar problem occurs with syntactically different array indices that may refer to the same array element).

For example, whether or not the formula $\texttt{o1.a} \doteq 1$ still holds after the execution of the assignment "`o2.a = 2;`" depends on whether or not `o1` and `o2` refer to the same object. Therefore, Java assignments cannot be symbolically executed by syntactic substitution, as done, for instance, in classical Hoare Logic. Solving this problem naively—by doing a case split—is inefficient and leads to heavy branching of the proof tree.

In the JavaDL calculus we use a different solution. It is based on the concept of *updates*, which can be seen as "semantic substitutions." Evaluating $\{loc := value\}\phi$ in a state is equivalent to evaluating ϕ in a modified state where *loc* evaluates to *value*, i.e., *loc* has been "semantically substituted" with *value* (see Section 3.4 for a discussion and a comparison of updates to assignments and substitutions).

The KeY system uses special simplification rules to compute the result of applying an update to terms and formulas that do not contain programs (see Section 3.4.2). Computing the effect of an update to a formula $\langle p\rangle\phi$ is delayed until p has been symbolically executed using other rules of the calculus. Thus, case distinctions are not only delayed but can often be avoided altogether, since (a) updates can be simplified *before* their effect has to be computed, and (b) their effect is computed when a maximal amount of information is available (namely *after* the symbolic execution of the whole program).

The basic assignment rule thus takes the following simple form:

$$\mathsf{assignment}\ \frac{\Longrightarrow \{loc := value\}\langle\pi\ \ \omega\rangle\phi}{\Longrightarrow \langle\pi\ \ loc\ \texttt{=}\ value\texttt{;}\ \ \omega\rangle\phi}$$

That is, it just turns the assignment into an update. Of course, this does not solve the problem of computing the effect of the assignment. This problem is postponed and solved later by the rules for simplifying updates.

Furthermore—and this is important—this "trivial" assignment rule is correct only if the expressions *loc* and *value* satisfy certain restrictions. The rule is only applicable if neither the evaluation of *loc* nor that of *value* can cause any side effects. Otherwise, other rules have to be applied first to analyze *loc* and *value*. For example, those other rules would replace the formula $\langle\texttt{x = ++i;}\rangle\phi$ with $\langle\texttt{i = i+1; x = i;}\rangle\phi$, before the assignment rule can be applied to derive first $\{\texttt{i} := \texttt{i+1}\}\langle\texttt{x = i;}\rangle\phi$ and then $\{\texttt{i} := \texttt{i+1}\}\{\texttt{x} := \texttt{i}\}\langle\rangle\phi$.

3.6.2 Rules for Handling General Assignments

In the following we use the notion *(program) location* to refer to local program variables, instance or static fields and array elements.

There are four classes of rules in the JavaDL calculus for treating general assignment expressions (that may have side-effects). These classes—corresponding to steps in the evaluation of an assignment—are induced by the evaluation order rules of Java:

1. Unfolding the left-hand side of the assignment.
2. Saving the location.
3. Unfolding the right-hand side of the assignment.
4. Generating an update.

Of particular importance is the fact that though the right-hand side of an assignment can change the variables appearing on the left-hand side, it cannot change the location scheduled for assignment, which is saved before the right-hand side is evaluated.

3.6.2.1 Step 1: Unfolding the Left-Hand Side

In this first step, the left-hand side of an assignment is unfolded if it is a nonsimple expression, i.e., if its evaluation may have side-effects. One of the following rules is applied depending on the form of the left-hand side expression. In general, these rules work by introducing a new local variable v_0, to which the value of a subexpression is assigned.

If the left-hand side of the assignment is a nonatomic field access—which is to say it has the form $nse\texttt{.}a$, where nse is a nonsimple expression—then the following rule is used:

$$\mathsf{assignmentUnfoldLeft}\ \frac{\Longrightarrow \langle \pi\ \ T_{nse}\ \ v_0\texttt{=}nse\texttt{;}\ \ v_0\texttt{.}a\texttt{=}e\texttt{;}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \ nse\texttt{.}a\texttt{=}e\texttt{;}\ \ \omega\rangle\phi}$$

Applying this rule yields an equivalent but simpler program, in the sense that the two new assignments have simpler left-hand sides, namely a local variable or an atomic field access.

Unsurprisingly, in the case of arrays, two rules are needed, since both the array reference and the index have to be treated. First, the array reference is analyzed:

$$\mathsf{assignmentUnfoldLeftArrayReference}$$
$$\frac{\Longrightarrow \langle \pi\ \ T_{nse}\ \ v_0\ \texttt{=}\ nse\texttt{;}\ \ v_0\texttt{[}e\texttt{]=}e_0\texttt{;}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \ nse\texttt{[}e\texttt{]=}e_0\texttt{;}\ \ \omega\rangle\phi}$$

Then, the rule for analyzing the array index can be applied:

$$\mathsf{assignmentUnfoldLeftArrayIndex}$$
$$\frac{\Longrightarrow \langle \pi\ \ T_v\ \ v_a\ \texttt{=}\ v\texttt{;}\ \ T_{nse}\ \ v_0\ \texttt{=}\ nse\texttt{;}\ \ v_a\texttt{[}v_0\texttt{]=}e\texttt{;}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \ v\texttt{[}nse\texttt{]=}e\texttt{;}\ \ \omega\rangle\phi}$$

3.6.2.2 Step 2: Saving the Location

After the left-hand side has been unfolded completely (i.e., has the form v, $v\texttt{.}a$ or $v\texttt{[}se\texttt{]}$), the right-hand side has to be analyzed. But before doing this, we have to memorize the location designated by the left-hand side. The reason is that the location affected by the assignment remains fixed even if evaluating the right-hand side of the assignment has a side effect changing the location to which the left-hand side points. For example, if `i` $\doteq 0$, then `a[i] = ++i;` has to update the location `a[0]` even though evaluating the right-hand side of the assignment changes the value of `i` to 1.

Since there is no universal "location" or "address-of" operator in Java, this memorizing looks different for different kinds of expressions appearing on the left. The choice here is between field and array accesses. For local variables, the memorizing step is not necessary, since the "location value" of a variable is syntactically defined and cannot be changed by evaluating the right-hand side.

We will start with the rule variant where a field access is on the left. It takes the following form; the components of the premiss are explained in Table 3.3:

$$\mathsf{assignmentSaveLocation}\quad \frac{\Longrightarrow \langle \pi\ \ memorize\,;\ \ unfoldr\,;\ \ update\,;\ \ \omega\rangle \phi}{\Longrightarrow \langle \pi\ \ v\,\texttt{.}\,a\texttt{=}nse\,;\ \ \omega\rangle \phi}$$

Table 3.3 Components of rule assignmentSaveLocation for field accesses $v\texttt{.}a\texttt{=}nse$

memorize	$T_v\ \ v_0\ \texttt{=}\ v\texttt{;}$	
unfoldr	$T_{nse}\ \ v_1\ \texttt{=}\ nse\texttt{;}$	set up Step 3
update	$v_0\texttt{.}a\ \texttt{=}\ v_1\texttt{;}$	set up Step 4

There is a very similar rule for the case where the left-hand side is an array access, i.e., the assignment has the form $v\texttt{[}se\texttt{]=}nse$. The components of the premiss for that case are shown in Table 3.4.

Table 3.4 Components of rule assignmentSaveLocation for array accesses $v\texttt{[}se\texttt{]=}nse$

memorize	$T_v\ \ v_0\ \texttt{=}\ v\texttt{;}\ T_{se}\ \ v_1\ \texttt{=}\ se\texttt{;}$	
unfoldr	$T_{nse}\ \ v_2\ \texttt{=}\ nse\texttt{;}$	set up Step 3
update	$v_0\texttt{[}v_1\texttt{]}\ \texttt{=}\ v_2\texttt{;}$	set up Step 4

[a] This includes an implicit test that v is not `null` when v`.length` is analyzed.

3.6.2.3 Step 3: Unfolding the Right-Hand Side

In the next step, after the location that is changed by the assignment has been memorized, we can analyze and unfold the right-hand side of the expression. There are several rules for this, depending on the form of the right-hand side. As an example, we give the rule for the case where the right-hand side is a field access $nse\texttt{.}a$ with a nonsimple object reference nse:

assignmentUnfoldRight

$$\frac{\Longrightarrow \langle \pi\ T_{nse}\ v_0\ \texttt{=}\ nse\texttt{;}\ v\ \texttt{=}\ v_0\texttt{.}a\texttt{;}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ v\ \texttt{=}\ nse\texttt{.}a\texttt{;}\ \omega\rangle\phi}$$

The case when the right-hand side is a method call is discussed in the section on method calls (Section 3.6.5).

3.6.2.4 Step 4: Generate an Update

The fourth and final step of treating assignments is to turn them into an update. If both the left- and the right-hand side of the assignment are simple expressions, the basic assignment rule applies:

$$\text{assignment}\ \frac{\Longrightarrow \{lhs := se^*\}\langle \pi\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ lhs\ \texttt{=}\ se\texttt{;}\ \omega\rangle\phi}$$

The value se^* appearing in the update is not identical to the se in the program because creating the update requires replacing any Java operators in the program expression se by their JavaDL counterparts in order to obtain a proper logical term. For example, the Java division operator `/` is replaced by the function symbol *javaDivInt* (or *javaDivLong* depending on the promoted type of its arguments). These function symbols are then further replaced according to the chosen integer semantics (see Section 5.4). The KeY system performs this conversion automatically to construct se^* from se. The complete list of predefined JavaDL operators is given in Appendix B.

If there is an atomic field access $v\texttt{.}a$ either on the left or on the right of the assignment, no further unfolding can be done and the possibility has to be considered here that the object reference may be `null`—which would result in a `NullPointerException`. Depending on whether the field access is on the left or on the right of the assignment one of the following rules applies:

assignment

$$\frac{\begin{array}{l} v \not\doteq \texttt{null} \Longrightarrow \{v_0 := select_A(\texttt{heap}, v, Class::\$a)\}\langle \pi\ \ \omega\rangle\phi \\ v \doteq \texttt{null} \Longrightarrow \langle \pi\ \texttt{throw new NullPointerException();}\ \omega\rangle\phi \end{array}}{\Longrightarrow \langle \pi\ v_0\ \texttt{=}\ v\texttt{.}a\texttt{;}\ \omega\rangle\phi}$$

assignment

$$\frac{\begin{array}{l} v \not\doteq \texttt{null} \Longrightarrow \{\texttt{heap} := store(\texttt{heap}, v, Class::\$a, se^*)\}\langle \pi\ \ \omega\rangle\phi \\ v \doteq \texttt{null} \Longrightarrow \langle \pi\ \texttt{throw new NullPointerException();}\ \omega\rangle\phi \end{array}}{\Longrightarrow \langle \pi\ v\texttt{.}a\ \texttt{=}\ se\texttt{;}\ \omega\rangle\phi}$$

In the rules you may have noticed that the field a is referred to by its unique field constant $Class::\$a$. This field constant unambiguously refers to the field named a of type A declared in the class $Class$ (where $Class$ is the fully qualified name). Determining $Class$ can be nontrivial, in particular in the presence of field hiding. Hiding occurs when derived classes declare fields with the same name as in the superclass. Inside a program the exact field reference can be determined from the short name a using the static type of the target expression and the program context, in which the reference appears. Since logical terms do not have a program context, hidden fields have to be immediately disambiguated by the assignment rule.

The KeY system's pretty-printer tries to improve readability of these terms by displaying the shorthand $v.a$ for $select_A(\mathtt{heap}, o, a)$, whenever the select expression is in a defined normalform and no hiding occurs (for a thorough description of pretty printing see Section 16.2). In the following, we use this shorthand notation unless there is a danger of confusion.

For array access, we have to consider the possibility of an `ArrayIndexOutOf-BoundsException` in addition to that of a `NullPointerException`. Thus, the rule for array access on the right of the assignment takes the following form (there is a slightly more complicated rule for array access on the left as it needs to account for `ArrayStoreExceptions`):

assignment

$$\frac{\begin{array}{l} v \neq \mathtt{null},\ se^* \geq 0,\ se^* < v.\mathtt{length} \Longrightarrow \\ \qquad \{v_0 := select_A(\mathtt{heap}, v, arr(se^*))\}\langle \pi\ \ \omega\rangle\phi \\ v \doteq \mathtt{null} \Longrightarrow \\ \qquad \langle \pi\ \mathtt{throw\ new\ NullPointerException();}\ \ \omega\rangle\phi \\ v \neq \mathtt{null},\ (se^* < 0 \lor se^* \geq v.\mathtt{length}) \Longrightarrow \\ \qquad \langle \pi\ \mathtt{throw\ new\ ArrayIndexOutOfBoundsException();}\ \ \omega\rangle\phi \end{array}}{\Longrightarrow \langle \pi\ \ v_0\ \mathtt{=}\ v\mathtt{[}se\mathtt{];}\ \ \omega\rangle\phi}$$

Please note that, if possible, KeY's pretty-printer uses the shorthand notation $v[se^*]$ for $select_A(\mathtt{heap}, v, arr(se^*))$; see Section 16.2.

The JVM throws exceptions such as the `ArrayIndexOutOfBoundsException` and the `NullPointerException` to signal an error condition during program execution. The assignment rules shown above faithfully model this behavior by introducing explicit `throw` statements during symbolic execution for those cases where the JVM would throw an exception.

However, the KeY system actually contains three user-selectable calculus variations for reasoning about such exceptions. The three variations are: *ban*, *allow*, and *ignore* (see Section 15.2.3 for an explanation of how to select different rule sets). The variation of assignment shown above is *allow*—it is both sound and complete. The variation *ban* requires to prove that no JVM-thrown exceptions can occur—it is sound but incomplete, as programs relying on catching such exceptions cannot be proved correct. The upside of *ban* is smaller proof size, as less symbolic execution is necessary. The third calculus variation is *ignore*; it makes the assumption that all operations succeed and neither checks for nor generates JVM-thrown exceptions. This variation is yet more efficient but neither sound nor complete.

A variability similar in spirit can be observed in the part of the calculus for reasoning about integer arithmetic (see Section 5.4.3).

Example 3.18. Consider the JavaDL formula

$$pre \rightarrow \langle \texttt{i = 0; try \{ o.a = null; i = 1; \}} \; \texttt{catch(Exception e) \{\}} \rangle post$$

The following table shows the differences in provability of this formula for different combinations of *pre* and *post* and different choices for exception handling in the calculus:

		provable		
pre	*post*	*allow*	*ban*	*ignore*
$\texttt{o} \doteq \texttt{null}$	$\texttt{i} \doteq 0$	Yes	No	No
$\texttt{o} \doteq \texttt{null}$	$\texttt{i} \doteq 1$	No	No	Yes
$\texttt{o} \not\doteq \texttt{null}$	$\texttt{i} \doteq 0$	No	No	No
$\texttt{o} \not\doteq \texttt{null}$	$\texttt{i} \doteq 1$	Yes	Yes	Yes

3.6.3 Rules for Conditionals

Most `if-else` statements have a nonsimple expression (i.e., one with potential side-effects) as their condition. In this case, we unfold it in the usual manner first. This is achieved by the rule

$$\mathsf{ifElseUnfold}\quad \frac{\Longrightarrow \langle \pi \; \texttt{boolean}\; v \;\texttt{=}\; nse\texttt{;}\; \texttt{if}\; \texttt{(}v\texttt{)}\; p \;\texttt{else}\; q \;\; \omega \rangle \phi}{\Longrightarrow \langle \pi \; \texttt{if}\; \texttt{(}nse\texttt{)}\; p \;\texttt{else}\; q \;\; \omega \rangle \phi}$$

where v is a fresh Boolean variable.

After dealing with the nonsimple condition, we will eventually get back to the `if-else` statement, this time with the condition being a variable and, thus, a simple expression. Now it is time to take on the case distinction inherent in the statement. That can be done using the following rule:

$$\mathsf{ifElseSplit}\quad \frac{\begin{array}{l} se^* \doteq TRUE \Longrightarrow \langle \pi \;\; p \;\; \omega \rangle \phi \\ se^* \doteq FALSE \Longrightarrow \langle \pi \;\; q \;\; \omega \rangle \phi \end{array}}{\Longrightarrow \langle \pi \; \texttt{if}\; \texttt{(}se\texttt{)}\; p \;\texttt{else}\; q \;\; \omega \rangle \phi}$$

While perfectly functional, this rule has several drawbacks. First, it unconditionally splits the proof, even in the presence of additional information. However, the program or the sequent may contain the explicit knowledge that the condition is true (or false). In that case, we want to avoid the proof split altogether. Second, after the split, the condition *se* appears on both branches, and we then have to reason about the same expression twice.

A different solution is the following rule that translates a program with an `if-else` statement into a conditional formula:

$$\mathsf{ifElse}\ \frac{\Longrightarrow \text{if}(se^* \doteq \mathit{TRUE})\,\text{then}\ \langle \pi\ \ p\ \ \omega\rangle\phi\ \text{else}\ \langle \pi\ \ q\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{if}\ \ (se)\ \ p\ \texttt{else}\ q\ \ \omega\rangle\phi}$$

Note that the if-then-else in the premiss of this rule is a logical and not a program language construct (Definition 3.3).

The ifElse rule solves the problems of the ifElseSplit rule described above. The condition *se* only has to be considered once. And if additional information about its truth value is available, splitting the proof can be avoided. If no such information is available, however, it is still possible to replace the propositional if-then-else operator with its definition, resulting in

$$((se^* \doteq \mathit{TRUE}) \rightarrow \langle \pi\ \ p\ \ \omega\rangle\phi) \quad \wedge \quad ((se^* \not\doteq \mathit{TRUE}) \rightarrow \langle \pi\ \ q\ \ \omega\rangle\phi)$$

and carry out a case distinction in the usual manner.

A problem that the above rule does not eliminate is the duplication of the code part ω. Its double appearance in the premiss means that we may have to reason about the same piece of code twice. Leino [2005] proposes a solution for this problem within a verification condition generator system. However, to preserve the advantages of a symbolic execution, the KeY system here sacrifices some efficiency for the sake of usability. And, fortunately, this issue is hardly ever limiting in practice.

The rule for the `switch` statement, which also is conditional and leads to case distinctions in proofs, is not shown here. It transforms a `switch` statement into a sequence of `if` statements.

3.6.4 Unwinding Loops

The following rule “unwinds” `while` loops.[2] Its application is the prerequisite for symbolically executing the loop body. Unfortunately, just unwinding a loop repeatedly is only sufficient for its verification if the number of loop iterations has a known upper bound. And it is only practical if that number is small (as otherwise the proof gets too big).

If the number of loop iterations is not bounded, the loop has to be verified using (a) induction or (b) an invariant rule (see Sections 3.7.2 and 9.4.2). If induction is used, the unwind rule is also needed as the loop has to be unwound once in the step case of the induction.

In case the loop body does not contain `break` or `continue` statements (which is the standard case), the following simple version of the unwind rule can be applied:

$$\mathsf{loopUnwind}\ \frac{\Longrightarrow \langle \pi\ \texttt{if}\ \ (e)\ \ \{\ p\ \texttt{while}\ \ (e)\ \ p\ \}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{while}\ \ (e)\ \ p\ \ \omega\rangle\phi}$$

[2] Occurrences of `for` loops, enhanced `for` loops, and `do-while` loops are transformed into `while` loops by means of dedicated rules.

Otherwise, in the general case where `break` and/or `continue` occur, the following more complex rule version has to be used:

loopUnwind
$$\frac{\Longrightarrow \langle \pi\ \texttt{if (}e\texttt{)}\ l'\texttt{:\{}\ l''\texttt{:\{}\ p'\ \texttt{\}}\ l_1\texttt{:}\ldots l_n\texttt{:while (}e\texttt{) \{}\ p\ \texttt{\} \}}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ l_1\texttt{:}\ldots l_n\texttt{:while (}e\texttt{) \{}\ p\ \texttt{\}}\ \omega\rangle\phi}$$

where

- l' and l'' are new labels,
- p' is the result of (simultaneously) replacing in p
  - every "`break` l_i" (for $1 \leq i \leq n$) and every "`break`" (with no label) that has the `while` loop as its target by "`break` l'," and
  - every "`continue` l_i" (for $1 \leq i \leq n$) and every "`continue`" (with no label) that has the `while` loop as its target by "`break` l''."

  (The target of a `break` or `continue` statement with no label is the loop that immediately encloses it.)

The label list $l_1\texttt{:}\ldots l_n\texttt{:}$ usually has only one element or is empty, but in general a loop can have more than one label.

In the "unwound" instance p' of the loop body p, the label l' is the new target for `break` statements and l'' is the new target for `continue` statements, which both had the `while` loop as target before. This results in the desired behavior: `break` abruptly terminates the whole loop, while `continue` abruptly terminates the current instance of the loop body.

A `continue` (with or without label) is never handled directly by a JavaDL rule, because it can only occur in loops, where it is always transformed into a `break` statement by the loop rules.

3.6.5 Replacing Method Calls by their Implementation

Symbolic execution deals with method invocations by syntactically replacing the call by the called implementation (verification via contracts is described in Section 3.7.1). To obtain an efficient calculus we have conservatively extended the programming language (see Section 3.2.3) with two additional constructs: a method body statement, which allows us to precisely identify an implementation, and a `method-frame` block, which records the receiver of the invocation result and marks the boundaries of the inlined implementation.

3.6.5.1 Evaluation of Method Invocation Expressions

The process of evaluating a method invocation expression (method call) within our JavaDL calculus consists of the following steps:

1. Identifying the appropriate method.
2. Computing the target reference.
3. Evaluating the arguments.
4. Locating the implementation (or throwing a `NullPointerException`).
5. Creating the method frame.
6. Handling the `return` statement.

Since method invocation expressions can take many different shapes, the calculus contains a number of slightly differing rules for every step. Also, not every step is necessary for every method invocation.

3.6.5.2 Step 1: Identify the Appropriate Method

The first step is to identify the appropriate method to invoke. This involves determining the right method signature and the class where the search for an implementation should begin. Usually, this process is performed by the compiler according to the (quite complicated) rules of the Java language specification and considering only static information such as type conformance and accessibility modifiers. These rules have to be considered as a background part of our logic, which we will not describe here though, but refer to the Java language specification instead. In the KeY system this process is performed internally (it does not require an application of a calculus rule), and the implementation relies on the Recoder metaprogramming framework to achieve the desired effect (Recoder is available at recoder.sourceforge.net).

For our purposes, we discern three different method invocation modes:

Instance or "virtual" mode. This is the most common mode. The target expression references an object (it may be an implicit `this` reference), and the method is not declared static or private. This invocation mode requires dynamic binding.

Static mode. In this case, no dynamic binding is required. The method to invoke is determined in accordance with the declared static type of the target expression and not the dynamic type of the object to which this expression may point. The static mode applies to all invocations of methods declared `static`. The target expression in this case can be either a class name or an object referencing expression (which is evaluated and then discarded). The static mode is also used for instance methods declared `private` (in which case the evaluated target reference is not discarded but used to identify the object on which to invoke the method).

Super mode. This mode is used to access the methods of the immediate superclass. The target expression in this case is the keyword `super`. The super mode bypasses any overriding declaration in the class that contains the method invocation.

Below, we present the rules for every step in a method invocation. We concentrate on the virtual invocation mode and discuss other modes only where significant differences occur.

3.6.5.3 Step 2: Computing the Target Reference

The following rule applies if the target expression of the method invocation is not a simple expression and may have side-effects. In this case, the method invocation gets unfolded so that the target expression can be evaluated first.

methodCallUnfoldTarget

$$\frac{\Longrightarrow \langle \pi\ T_{nse}\ v_0\ \texttt{=}\ nse\texttt{;}\ lhs\ \texttt{=}\ v_0\texttt{.}mname\texttt{(}args\texttt{);}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ lhs\ \texttt{=}\ nse\texttt{.}mname\texttt{(}args\texttt{);}\ \ \omega\rangle\phi}$$

This step is not performed if the target expression is the keyword `super` or a class name. For an invocation of a static method via a reference expression, this step *is* performed, but the result is discarded later on.

3.6.5.4 Step 3: Evaluating the Arguments

If a method invocation has arguments that need to be evaluated, i.e., if at least one of the arguments is not a simple expression, then the arguments have to be evaluated before control is transferred to the method body. This is achieved by the following rule:

methodCallUnfoldArguments

$$\frac{\begin{array}{l}\Longrightarrow \langle \pi\ T_{e_1}\ a_1\texttt{=}e_1\texttt{;}\ \ \ldots\texttt{;}\ \ T_{e_n}\ \ a_n\texttt{=}e_n\texttt{;}\\ \qquad lhs\ \texttt{=}\ se\texttt{.}mname\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{);}\\ \quad \omega\rangle\phi\end{array}}{\Longrightarrow \langle \pi\ lhs\ \texttt{=}\ se\texttt{.}mname\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{);}\ \ \omega\rangle\phi}$$

The rule unfolds the arguments using fresh variables in the usual manner.

In the *instance* invocation mode, the target expression *se* must be simple (otherwise the rules from Step 2 apply). Furthermore, argument evaluation has to happen even if the target reference is `null`, which is not checked until the next step.

3.6.5.5 Step 4: Locating the Implementation

This step has two purposes in our calculus: to bind the argument values to the formal parameters and to simulate dynamic binding (for *instance* invocations). Both are achieved with the following rule:

methodCall

$$\frac{\begin{array}{l} se \not\doteq \texttt{null} \Longrightarrow \langle \pi\ T_{lhs}\ v_0\texttt{;}\ paramDecl;\ ifCascade;\ lhs\ \texttt{=}\ v_0\texttt{;}\ \omega\rangle\phi \\ se \doteq \texttt{null} \Longrightarrow \langle \pi\ \texttt{throw new NullPointerException();}\ \omega\rangle\phi \end{array}}{\Longrightarrow \langle \pi\ lhs\ \texttt{=}\ se\texttt{.}mname\texttt{(}se_1\texttt{,}\ldots\texttt{,}se_n\texttt{);}\ \omega\rangle\phi}$$

The code piece *paramDecl* introduces and initializes new local variables that later replace the formal parameters of the method. That is, *paramDecl* abbreviates

$$T_{se_1}\ p_1\ \texttt{=}\ se_1\texttt{;}\ \ldots\ T_{se_n}\ p_n\ \texttt{=}\ se_n\texttt{;}$$

The code schema *ifCascade* simulates dynamic binding. Using the signature of *mname*, we extract the set of classes that implement this particular method from the given Java program. Due to the possibility of method overriding, there can be more than one class implementing a particular method. At runtime, an implementation is picked based on the dynamic type of the target object—a process known as dynamic binding. In our calculus, we have to do a case distinction as the dynamic type is in general not known. We employ a sequence of nested `if` statements that discriminate on the type of the target object, cast the callee variable to the static type in which the method implementation is found, and refer to the distinct method implementations via method body statements (see Section 3.2.3). Thus, *ifCascade* abbreviates:

```
if (se instanceof C1) {
   C1 target = (C1)se;   v0 = target.mname(p1,...,pn)@C1;
} else if (se instanceof C2) {
   C2 target = (C2)se;   v0 = target.mname(p1,...,pn)@C2;
⋮
} else if (se instanceof Ck-1) {
   Ck-1 target = (Ck-1)se;   v0 = target.mname(p1,...,pn)@Ck-1;
else {
   Ck target = (Ck)se;   v0 = target.mname(p1,...,pn)@Ck;
}
```

The order of the if statements is a bottom-up latitudinal search over all classes $C_1,\ldots,C_k$ of the class inheritance tree that implement *mname*`(...)`. In other words, the more specialized classes appear closer to the top of the cascade. Formally, if $i < j$ then $C_j \not\sqsubseteq C_i$.

If the invocation mode is *static* or *super* no *ifCascade* is created. The single appropriate method body statement takes its place. Furthermore, the check whether *se* is `null` is omitted in these modes, though not for private methods.

Please note that this step in method invocation and its associated rule forfeit modular correctness: The rule is only sound if the constructed if-cascade is complete, which requires all relevant methods to be known at the time of rule application (see Section 9.1.3).

3.6.5.6 Step 5: Creating the Method Frame

In this step, the method body statement v_0=*se*.*mname*(...)@*Class* is replaced by the implementation of *mname* from the class *Class* and the implementation is enclosed in a method frame:

$$\mathsf{methodBodyExpand}\quad \frac{\Longrightarrow \langle\pi\ \texttt{method-frame(result->}\mathit{lhs}\texttt{,}\ \texttt{source=}\mathit{mname}\texttt{(}T_1,\ldots,T_n\texttt{)@}\mathit{Class}\texttt{,}\ \texttt{this=}\mathit{se}\ \texttt{) : \{}\ \mathit{body}\ \texttt{\}}\ \omega\rangle\phi}{\langle\pi\ \mathit{lhs}\texttt{=}\mathit{se}\texttt{.}\mathit{mname}\texttt{(}v_1,\ldots,v_n\texttt{)@}\mathit{Class}\texttt{;}\ \omega\rangle\phi \Longrightarrow}$$

in the implementation *body* the formal parameters of types $T_1,\ldots,T_n$ of *mname* are syntactically replaced by $v_1,\ldots,v_n$.

3.6.5.7 Step 6: Handling the `return` Statement

The final stage of handling a method invocation, after the method body has been symbolically executed, involves committing the return value (if any) and transferring control back to the caller. We postpone the description of treating method termination resulting from an exception (as well as the intricate interaction between a `return` statement and a `finally` block) until the following section on abrupt termination.

The basic rule for the `return` statement is:

$$\mathsf{methodCallReturn}\quad \frac{\Longrightarrow \langle\pi\ \texttt{method-frame(...):\{}\ v\texttt{=}\mathit{se}\texttt{;}\ \texttt{\}}\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ \texttt{method-frame(result->}v\texttt{, ...) : \{ return}\ \mathit{se}\texttt{;}\ p\ \texttt{\}}\ \omega\rangle\phi}$$

We assume that the return value has already undergone the usual unfolding analysis, and is now a simple expression *se*. Now, we need to assign it to the right variable *v* within the invoking code. This variable is specified in the head of the method frame. A corresponding assignment is created and *v* disappears from the method frame. Any trailing code *p* is also discarded.

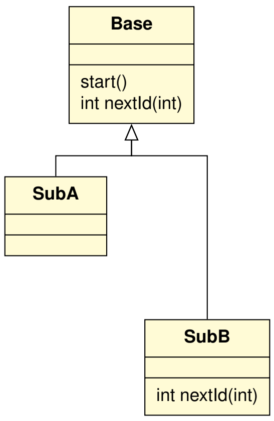
After the assignment of the return value is symbolically executed, we are left with an empty method frame, which can now be removed altogether. This is achieved with the rule

$$\mathsf{methodCallEmpty}\quad \frac{\Longrightarrow \langle\pi\quad \omega\rangle\phi}{\Longrightarrow \langle\pi\ \texttt{method-frame(...) : \{ \}}\ \omega\rangle\phi}$$

In case the method is void or if the invoking code simply does not assign the value of the method invocation to any variable, this fact is reflected by the variable *v* missing from the method frame. Then, slightly simpler versions of the return rule are used, which do not create an assignment.

3.6.5.8 Example for Handling a Method Invocation

Consider the example program from Figure 3.1. The method `nextId()` returns for a given integer value `id` some next available value. In the `Base` class this method is implemented to return `id+1`. The class `SubA` inherits and retains this implementation. The class `SubB` overrides the method to return `id+2`, which is done by increasing the result of the implementation in `Base` by one.



```
public class Base {
  public int nextId(int i) {
    return ++i;
  }
}

public class SubA extends Base {
}

public class SubB extends Base {
  public int nextId(int i) {
    return super.nextId(i)+1;
  }
}
```

Figure 3.1 An example program with method overriding

We now show step by step how the following code, which invokes the method `nextId()` on an object of type `SubB`, is symbolically executed:

—— Java ——

```
Base o = new SubB();
res = o.nextId(i);
```

—— Java ——

First, the instance creation is handled, after which we are left with the actual method call. The effect of the instance creation is reflected in the updates attached to the formula, which we do not show here. Since the target reference `o` is already *simple* at this point, we skip Step 2. The same applies to the arguments of the method call and Step 3. We proceed with Step 4, applying the rule methodCall. This gives us two branches. One corresponds to the case where `o` is `null`, which can be discharged using the knowledge that `o` points to a freshly created object. The other branch assumes that `o` is not `null` and contains a formula with the following Java code (in the following, program part A is transformed into A′, B into B′ etc.):

—— Java ——
```
int j; {                                        (A)
    int i_1 = i;
    if (o instanceof SubB) {
        SubB target = (SubB)o;
        j=target.nextId(i_1)@SubB;
    } else {
        Base target = (Base)o;
        j=target.nextId(i_1)@Base;
    }
}
res=j;
```
—— Java ——

After dealing with the variable declarations, we reach the if-cascade simulating dynamic binding. In this case we happen to know the dynamic type of the object referenced by `o`. This eliminates the choice and leaves us with assigning `o` to a variable of the same static type where the implementation is been found, and finally, the method body statement pointing to the implementation from `SubB`:

—— Java ——
```
SubB target = (SubB)o;     (A')
j=target.nextId(i_1)@SubB;
res=j;
```
—— Java ——

After executing the variable declaration of `target` and assigning it the value of `o` (the cast succeeds because of the if-statement guard in the previous step), it is time for Step 5: unfolding the method body statement and creating a method frame. This is achieved by the rule methodBodyExpand:

—— Java ——
```
method-frame(result->j,source=nextId(int)@SubB,this=target):{ (A")
    return super.nextId(i_1)+1;    (B)
}
res=j;
```
—— Java ——

The method implementation has been inlined above. We start to execute it symbolically, unfolding the expression in the `return` statement in the usual manner, which gives us after some steps:

—— Java ——
```
method-frame(result->j, source=nextId(int)@SubB, this=target):{
    int j_2 = super.nextId(i_1); Ⓒ    (B')
    j_1=j_2+1;
    return  j_1;
}
res=j;
```
—— Java ——

The active statement is now again a method invocation, this time with the `super` keyword. The method invocation process starts again from scratch. Steps 2 and 3 can be omitted for the same reasons as above. Step 4 gives us the following code. Note that there is no if-cascade, since no dynamic binding needs to be performed.

—— Java ——
```
method-frame(result->j, source=nextId(int)@SubB, this=target):{
    int j_3; {                                (C')
        int i_2 = i_1;
        j_3=target.nextId(i_2)@Base;
    }
    j_2=j_3;
    j_1=j_2+1;
    return  j_1;
}
res=j;
```
—— Java ——

Now it is necessary to remove the declarations and perform the assignments to reach the method body statement `j_3=target.nextId(i_2)@Base;`. Then, this statement can be unpacked (Step 5), and we obtain two nested method frames. The second method frame retains the value of `this`, while the implementation source is now taken from the superclass:

```
method-frame(result->j, source=nextId(int)@SubB, this=target):{
    method-frame(result->j_3,                                  C"
                 source=nextId(int)@Base, this=target)  : {
        return ++i_2;  D
    }
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```

The return expression is unfolded until we arrive at a simple expression. The actual return value is recorded in the updates attached to the formula. The code in the formula then is:

```
method-frame(result->j, source=nextId(int)@SubB, this=target):{
    method-frame(result->j_3,                                E
                 source=nextId(int)@Base, this=target) : {
           return j_4;  D'
    }
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```

Now we can perform Step 6 (rule methodCallReturn), which replaces the `return` statement of the inner method frame with the assignment to the variable `j_3`. We know that `j_3` is the receiver of the return value, since it was identified as such by the method frame (this information is removed with the rule application).

Java
```
method-frame(result->j, source=nextId(int)@SubB, this=target):{
    method-frame(source=nextId(int)@Base, this=target) : { E'
        j_3=j_4;
    }
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```
Java

The assignment `j_3=j_4;` can be executed as usual, generating an update, and we obtain an empty method frame.

Java
```
method-frame(result->j, source=nextId(int)@SubB, this=target):{
    method-frame(source=nextId(int)@Base, this=target):{  E''
    }
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```
Java

The empty frame can be removed with the rule methodCallEmpty, completing Step 6. The invocation depth has now decreased again. We obtain the program:

Java
```
method-frame(result->j, source=nextId(int)@SubB, this=target):{
    j_2=j_3;
    j_1=j_2+1;
    return j_1;
}
res=j;
```
Java

From here, the execution continues in an analogous manner. The outer method frame is eventually removed as well.

3.6.6 Instance Creation and Initialization

In this section we cover the process of instance creation and initialization. We do not go into details of array creation and initialization, since it is sufficiently similar.

3.6.6.1 Instance Creation and the Constant Domain Assumption

JavaDL, like many modal logics, operates under the technically useful constant domain semantics (all program states have the same universe). This means, however, that all instances that are ever created in a program have to exist a priori. To resolve this seeming paradox, we introduce *implicit fields* that allow to change and query the program-visible instance state (created, initialized, etc.); see Table 3.5. These implicit fields behave as the usual class or instance attributes, except that they are not declared by the user but by the logic designer. To distinguish them from normal (user-declared) fields, their names are enclosed in angled brackets.

Table 3.5 Implicit object repository and status fields

Modifier	Implicit field	Declared in	Explanation
`protected`	`boolean <created>`	`Object`	indicates whether the object has been created
`protected`	`boolean <initialised>`	`Object`	indicates whether the object has been initialized

Example 3.19. To express that the field `head` declared in some class `A` is nonnull for all created and initialized objects of type `A`, one can use the following formula:

$$\forall a : A.(a \not\doteq \texttt{null} \land a.\texttt{<created>} \doteq \mathit{TRUE} \ \rightarrow\ (a.\texttt{head} \not\doteq \texttt{null}))$$

In future, we use the easier to read *created* to refer to the field `<created>`, except for syntax used as part of KeY input files or similar.

3.6.6.2 Overview of the Java Instance Creation and Initialization Process

We use an approach to handle instance creation and initialization that is based on program transformation. The transformation reduces a Java program p to a program p' such that the behavior of p (with initialization) is the same as that of p' when initialization is disregarded. This is done by inserting code into p that explicitly executes the initialization.

The transformation inserts code for explicitly executing all initialization processes. To a large extent, the inserted code works by invoking implicit class or instance methods (similar to implicit fields), which do the actual work. An overview of all implicit methods introduced is given in Table 3.6.

Table 3.6 Implicit methods for object creation and initialization declared in every nonabstract type *T* (syntactic conventions from Figure 3.2)

Static methods	
`public static` *T* `<createObject>()`	main method for instance creation and initialisation
`private static` *T* `<allocate>()`	allocation of an unused object from the object repository
Instance methods	
`protected void <prepare>()`	assignment of default values to all instance fields
mods T `<init>(`*params*`)`	execution of instance initializers and the invoked constructor

The transformation covers all details of initialization in Java, except that we only consider nonconcurrent programs and no reflection facilities (in particular no instances of `java.lang.Class`). Initialization of classes and interfaces (also known as static initialization) is fully supported for the single threaded case. KeY passes the static initialization challenge stated by Jacobs et al. [2003].

In the following, we use the schematic class form shown in Figure 3.2.

```
mods_0 class T {
  mods_1 T_1 a_1 = initExpression_1;
  ⋮
  mods_m T_m a_m = initExpression_m;

  {
    initStatement_{m+1};
    ⋮
    initStatement_l;
  }

  mods T(params) {
     st_1;
     ⋮
     st_n;
  }
  ⋮
}
```

Figure 3.2 Initialization part in a schematic class

Example 3.20. Figure 3.3 shows a class `Person` and its mapping to the schematic class declaration of Figure 3.2. There is only one initializer statement in class `Person`, namely "`id = 0`," which is induced by the corresponding field declaration of `id`.

```
class Person {
  private int id = 0;

  public Person(int persID) {
    id = persID;
  }
}
```

$mods_0$	$\mapsto$ –
T	$\mapsto$ `Person`
$mods_1$	$\mapsto$ `private`
T_1	$\mapsto$ `int`
a_1	$\mapsto$ `id`
$initExpression_1$	$\mapsto$ `0`
$mods$	$\mapsto$ `public`
$params$	$\mapsto$ `int persID`
st_1	$\mapsto$ `id = persID`

Figure 3.3 Example for the mapping of a class declaration to the schema of Figure 3.2

To achieve a uniform presentation we also stipulate that:

1. The default constructor `public` T`()` exists in T in case no explicit constructor has been declared.
2. Unless $T =$ `Object`, the statement st_1 must be a constructor invocation. If this is not the case in the original program, "`super();`" is added explicitly as the first statement.

Both of these conditions reflect the actual semantics of Java.

3.6.6.3 The Rule for Instance Creation and Initialization

The instance creation rule

$$\mathsf{instanceCreation}\ \frac{\begin{array}{l}\Longrightarrow \langle \pi\ T\ v_0\ \texttt{=}\ T\texttt{.<createObject>();}\\ \quad T_1\ a_1\ \texttt{=}\ e_1\texttt{;}\ \ \ldots\texttt{;}\ T_1\ a_n\ \texttt{=}\ e_n\texttt{;}\\ \quad v_0\texttt{.<init>(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{)@T;}\\ \quad v_0\texttt{.<initialised> = true;}\\ \quad v\ \texttt{=}\ v_0\texttt{;}\\ \omega\rangle\phi\end{array}}{\Longrightarrow \langle \pi\ v\ \texttt{= new}\ T\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{);}\ \omega\rangle\phi}$$

replaces an instance creation expression "v `= new` T`(`e_1`,...,`e_n`)`" by a sequence of statements. The implicit static method `<createObject>()`is declared in each nonabstract class T as follows:

```
public static T <createObject>() {
   T newObject = T.<allocate>();
         // Invoke the preparation method to assign default values to
         // instance fields
   newObject.<create>();
         // Return the newly created object in order to initialize it:
   return newObject;
}
```

`<createObject>()` delegates its work to a series of other helper methods. The generated code can be divided into three phases, which we examine in detail below:

1. `<allocate>()`: Allocate space on the heap, mark the object as created (as explained above, it is not really "created"), and assign the reference to a temporary variable v_0.
2. `<create>()`: Prepare the object by assigning all fields their default values.
3. `<init>()`: Initialize the object and subsequently mark it as initialized. Note that the rule uses the method body statement instead of a normal method invocation. This is possible as we exactly know which constructor has been invoked and it allows us to achieve an improved performance as we do not need to use dynamic dispatch.

The reason for assigning v_0 to v in the last step is to ensure correct behavior in case initialization terminates abruptly due to an exception.[3]

3.6.6.4 Phase 1: Instance Allocation: `<allocate>`

During the first phase, an implicit method called `<allocate>()`, performs the central interaction with the heap. The `<allocate>()` method has no Java implementation; its semantics is given by the following rule instead:

allocateInstance

$$\frac{\begin{array}{l} o' \not\doteq \mathtt{null},\ \mathit{exactInstance}_T(o') \doteq \mathit{TRUE}, \\ \big(\mathit{wellFormed}(\mathtt{heap}) \to \mathit{select}_{boolean}(\mathtt{heap}, o', \mathit{created}) \doteq \mathit{FALSE}\big) \\ \Longrightarrow \{\mathtt{heap} := \mathit{create}(\mathtt{heap}, o')\} \\ \quad \{\mathit{lhs} := o'\} \\ \quad\quad \langle \pi\ \ \omega \rangle \phi \end{array}}{\Longrightarrow \langle \pi\ \ \mathit{lhs}\ \mathtt{=}\ T\mathtt{.<allocate>();}\ \ \omega \rangle \phi}$$

where $o' : T \in \mathrm{FSym}$ is a fresh symbol

[3] Java does not prevent creating and accessing partly initialized objects. This can be done, for example, by assigning the object reference to a static field during initialization. This behavior is modeled faithfully in the calculus. In such cases the preparation phase guarantees that all fields have a definite value.

The rule introduces a *fresh* constant symbol o' to represent the new object, i.e., a constant symbol not occurring anywhere in the conclusion. The rule adds three assumptions about the otherwise unknown object represented by o': (i) it is different from *null*; (ii) its dynamic type is T; and (iii) if the heap is well-formed, then the object is not yet created. These assumptions are always satisfiable, because there is an infinite reservoir of objects of every type, and because in a well-formed heap only a finite number of them is created.

The new object is then marked as "created" by setting its *created* field to true, and the reference to the newly created object is assigned to the program variable *lhs*.

3.6.6.5 Phase 2: Preparation: `<create>`

During the second phase, an implicit method called `<create>` marks the object as not yet initialized (`this.<initialized>=false;`) and calls the implicit method `<prepare>()`, which makes sure that all fields, including the ones declared in the superclasses, are assigned their default values.[4] Up to this point no user code is involved, which ensures that all field accesses by the user observe a definite value. This value is given by the function *defaultValue* that maps each type to its default value (e.g., `int` to 0). The concrete default values are specified in the Java language specification [Gosling et al., 2013, § 4.5.5]. The method `<prepare>()` used for preparation is shown in Figure 3.4.[5]

```
protected void <prepare>() {
            // Prepare the fields declared in the superclass...
     super.<prepare>();              // unless T = Object
            // Then assign each field a_i of type T_i declared in T
            // to its default value:
       a_1 = defaultValue(T_1);
       ...
       a_m = defaultValue(T_m);
}
```

Figure 3.4 Implicit method `<prepare>()`

[4] Since class declarations are given beforehand this is possible with a simple enumeration. In case of arrays, a quantified update is used to achieve the same effect, even when the actual array size is not known.

[5] In the KeY system, `<create>()` does not call `<prepare>()` on the new object directly. Instead it invokes another implicitly declared method called `<prepareEnter>()`, which has private access and whose body is identical to the one of `<prepare>()`. The reason is that due to the `super` call in `<prepare>()`'s body, its visibility must be at least `protected` such that a direct call would trigger dynamic method dispatching, which is unnecessary and would lead to a larger proof.

3.6.6.6 Instance Initialization: `<init>`

After the preparation of the new object, the user-defined initialization code can be processed. Such code can occur

- as a field initializer expression "*T attr* `=` *val*`;`" (e.g., `(*)` in Figure 3.5); the corresponding initializer statement is *attr* `=` *val*;
- as an instance initializer block (similar to `(**)` in Figure 3.5); such a block is also an initializer statement;
- within a constructor body (like `(***)` in Figure 3.5).

```
class A {                                             private <init>() {
  (*)   private int a = 3;                              super.<init>();
  (**)  {a++;}                                          a = 3;
        public int b;                        ........>  {a++;}
                                                        a = a + 2;
  (***) private A() {                                 }
          a = a + 2;
        }                                             public <init>(int i) {
  (***) public A(int i) {                               this.<init>();
          this();                            ........>  a = a + i;
          a = a + i;                                  }
        }                                           }
  ...
```

Figure 3.5 Example for constructor normal form

For each constructor *mods T*`(`*params*`)` of *T* we provide a constructor normal form *mods T* `<init>(`*params*`)`, which includes (1) the initialization of the superclass, (2) the execution of all initializer statements in source code order, and finally (3) the actual constructor body. In the initialization phase the arguments of the instance creation expression are evaluated and passed on to this constructor normal form. An example of the normal form is given in Figure 3.5.

The exact blueprint for building a constructor normal form is shown in Figure 3.6, using the conventions of Figure 3.2. Due to the uniform class form assumed above, the first statement st_1 of every original constructor is either an alternate constructor invocation or a superclass constructor invocation (with the notable exception of $T =$ `Object`). Depending on this first statement, the normal form of the constructor is built to do one of two things:

1. $st_1 =$ `super(`*args*`)`: Recursive restart of the initialization phase for the superclass of T. If $T =$ `Object` stop. Afterwards, initializer statements are executed in source code order. Finally, the original constructor body is executed.
2. $st_1 =$ `this(`*args*`)`: Recursive restart of the initialization phase with the alternate constructor. Afterwards, the original constructor body is executed.

If one of the above steps fails, the initialization terminates abruptly throwing an exception.

```
mods T <init>(params) {
          // invoke constructor
          // normal form of superclass
          // (only if T ≠ Object)
     super.<init>(args);

          // add the initializer
          // statements:
     initStatement_1;
     ...
     initStatement_l;
          // append constructor body
     st_s; ... st_n;
          // if T = Object then s = 1
          // otherwise s = 2
}
```

(a) st_1 = `super(`*args*`)` in the original constructor

```
mods T <init>(params) {

          // constructor normal form
          // instead of this(args)
     this.<init>(args);
          // no initializer statements
          // if st_1 is an explicit
          // this() invocation



          // append constructor body
     st_2; ... st_n;
          // starting with its second
          // statement
}
```

(b) st_1 = `this(`*args*`)` in the original constructor

Figure 3.6 Building the constructor normal form

3.6.7 Handling Abrupt Termination

3.6.7.1 Abrupt Termination in JavaDL

In Java, the execution of a statement can terminate *abruptly* (besides terminating normally and not terminating at all). Possible reasons for an abrupt termination are (a) that an exception has been thrown, (b) that a statement (usually a loop or a `switch`) is terminated with `break`, (c) that a single loop iteration is terminated with the `continue` statement, and (d) that the execution of a method is terminated with the `return` statement. Abrupt termination of a statement either leads to a redirection of the control flow after which the program execution resumes (for example, if an exception is caught), or the whole program terminates abruptly (if an exception is not caught).

Note, that the KeY system contains three user-selectable calculus variations for reasoning about run-time exceptions that may be thrown by the JVM (e.g., `NullPointerException`); see Section 3.6.2 and 15.2.3.

3.6.7.2 Evaluation of Arguments

If the argument of a `throw` or a `return` statement is a nonsimple expression, the statement has to be unfolded first such that the argument can be (symbolically) evaluated:

$$\mathsf{throwEvaluate}\ \frac{\Longrightarrow \langle \pi\ T_{nse}\ v_0\ \texttt{=}\ nse\texttt{;}\ \texttt{throw}\ v_0\texttt{;}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{throw}\ nse\texttt{;}\ \omega\rangle\phi}$$

3.6.7.3 If the Whole Program Terminates Abruptly

In JavaDL, an *abruptly* terminating statement—where the abrupt termination does not just change the control flow but actually terminates the whole program p in a modal operator $\langle p \rangle$ or $[p]$—has the same semantics as a *nonterminating* statement (Definition 3.5). For that case rules such as the following are provided in the JavaDL calculus for all abruptly terminating statements:

throwDiamond
$$\frac{\Longrightarrow \mathrm{false}}{\Longrightarrow \langle \texttt{throw}\ se\texttt{;}\ \omega \rangle \phi}$$

throwBox
$$\frac{\Longrightarrow \mathrm{true}}{\Longrightarrow [\texttt{throw}\ se\texttt{;}\ \omega] \phi}$$

Note, that in these rules, there is no inactive prefix π in front of the `throw` statement. Such a π could contain a `try` with accompanying `catch` clause that would catch the thrown exception. However, the rules throwDiamond, throwBox etc. must only be applied to uncaught exceptions. If there is a prefix π, other rules described below must be applied first.

3.6.7.4 If the Control Flow is Redirected

The case where an abruptly terminating statement does not terminate the whole program in a modal operator but only changes the control flow is more difficult to handle and requires more rules. The basic idea for handling this case in our JavaDL calculus are rules that *symbolically* execute the change in control flow by syntactically rearranging the affected program parts.

The calculus rules have to consider the different combinations of prefix-context (beginning of a block, method-frame, or `try`) and abruptly terminating statement (`break`, `continue`, `return`, or `throw`). Below, rules for all combinations are discussed—with the following exceptions:

- The rule for the combination method frame/`return` is part of handling method invocations (Step 6 in Section 3.6.5.1).
- Due to restrictions of the Java language specification, the combination method frame/`break` does not occur.
- Since the `continue` statement can only occur within loops, all occurrences of `continue` are handled by the loop rules.

Moreover, `switch` statements, which may contain a `break`, are not considered here; they are transformed into a sequence of `if` statements.

3.6.7.5 Rule for Method Frame and `throw`

In this case, the method is terminated, but no return value is assigned. The `throw` statement remains unchanged (i.e., the exception is handed up to the invoking code):

$$\mathsf{methodCallThrow}\ \frac{\Longrightarrow \langle \pi\ \texttt{throw}\ se\texttt{;}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{method-frame(...) : \{throw}\ se\texttt{;}\ p\ \texttt{\}}\ \omega\rangle\phi}$$

3.6.7.6 Rules for `try` and `throw`

The following rule allows us to handle `try-catch-finally` blocks and the `throw` statement:

$$\mathsf{tryCatchFinallyThrow}$$
$$\frac{\begin{array}{l}\Longrightarrow \langle \pi\ \texttt{if (}se\texttt{ == null) \{}\\ \qquad\qquad \texttt{try \{ throw new NullPointerException(); \}}\\ \qquad\qquad \texttt{catch (}T\ v\texttt{) \{ }q\texttt{ \} }cs\texttt{ finally \{ }r\texttt{ \}}\\ \qquad\quad \texttt{\} else if (}se\texttt{ instanceof }T\texttt{) \{}\\ \qquad\qquad \texttt{try \{ }T\ v\texttt{; }v\texttt{ = (}T\texttt{)}se\texttt{; }q\texttt{ \} finally \{ }r\texttt{ \}}\\ \qquad\quad \texttt{\} else \{}\\ \qquad\qquad \texttt{try \{ throw }se\texttt{; \} }cs\texttt{ finally \{ }r\texttt{ \}}\\ \qquad\quad \texttt{\}}\\ \qquad \omega\rangle\phi\end{array}}{\begin{array}{l}\Longrightarrow \langle \pi\ \texttt{try \{ throw }se\texttt{; }p\texttt{\}}\\ \qquad\quad \texttt{catch (}T\ v\texttt{) \{ }q\texttt{ \} }cs\texttt{ finally \{ }r\texttt{ \}}\\ \qquad \omega\rangle\phi\end{array}}$$

The schema variable *cs* represents a (possibly empty) sequence of catch clauses. The rule covers three cases corresponding to the three cases in the premiss:

1. The argument of the `throw` statement is the null pointer (which, of course, in practice should not happen). In that case everything remains unchanged except that a `NullPointerException` is thrown instead of *null*.
2. The first catch clause catches the exception. Then, after binding the exception to v, the code q from the catch clause is executed.
3. The first catch clause does *not* catch the exception. In that case the first clause gets eliminated. The same rule can then be applied again to check further clauses.

Note, that in all three cases the code p after the `throw` statement gets eliminated.

When all catch clauses have been checked and the exception has still not been caught, the following rule applies:

$$\mathsf{tryFinallyThrow}$$
$$\frac{\begin{array}{l}\Longrightarrow \langle \pi\quad \texttt{if (}se\texttt{ == null) \{ }v_{se}\texttt{ = new NullPointerException(); \}}\\ \qquad\quad \texttt{else}\qquad\qquad\quad \texttt{\{ }v_{se}\texttt{ = }se\texttt{; \}}\\ \qquad\quad r\\ \qquad\quad \texttt{throw }v_{se}\texttt{;}\\ \quad \omega\rangle\phi\end{array}}{\Longrightarrow \langle \pi\ \texttt{try \{ throw }se\texttt{; }p\texttt{ \} finally \{ }r\texttt{ \}}\rangle\phi}$$

This rule moves the code *r* from the finally block to the front. The `try`-block gets eliminated so that the thrown exception now may be caught by other `try` blocks in π (or remain uncaught). The value of *se* has to be saved in v_{se} before the code *r* is executed as *r* might change *se*.

There is also a rule for `try` blocks that have been symbolically executed without throwing an exception and that are now empty and terminate normally (similar rules exist for empty blocks and empty method frames). Again, *cs* represents a finite (possibly empty) sequence of catch clauses:

$$\text{tryEmpty}\ \frac{\Longrightarrow \langle \pi\ \ r\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{try\{ \}}\ cs\ \texttt{\{}\ q\ \texttt{\}}\ \texttt{finally \{}\ r\ \texttt{\}}\ \omega\rangle\phi}$$

3.6.7.7 Rules for `try`/`break` and `try`/`return`

A `return` or a `break` statement within a `try-catch-finally` statement causes the immediate execution of the `finally` block. Afterwards the `try` statement terminates abnormally with the `break` or the `return` statement (a different abruptly terminating statement that may occur in the `finally` block takes precedence). This behavior is simulated by the following two rules (here, also, *cs* is a finite, possibly empty sequence of catch clauses):

tryBreak

$$\frac{\Longrightarrow \langle \pi\ \ r\ \texttt{break}\ l\texttt{;}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{try\{ break}\ l\texttt{;}\ p\ \texttt{\}}\ cs\ \texttt{\{}\ q\ \texttt{\}}\ \texttt{finally\{}\ r\ \texttt{\}}\ \omega\rangle\phi}$$

tryReturn

$$\frac{\Longrightarrow \langle \pi\ \ T_{v_r}\ v_0\ \texttt{=}\ v_r\texttt{;}\ r\ \texttt{return}\ v_0\texttt{;}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{try\{ return}\ v_r\texttt{;}\ p\ \texttt{\}}\ cs\ \texttt{\{}\ q\ \texttt{\}}\ \texttt{finally\{}\ r\ \texttt{\}}\ \omega\rangle\phi}$$

3.6.7.8 Rules for block/`break`, block/`return`, and block/`throw`

The following two rules apply to blocks being terminated by a `break` statement that does not have a label, or by a break statement with a label *l* identical to one of the labels $l_1,\ldots,l_k$ of the block ($k \geq 1$).

$$\text{blockBreakNoLabel}\ \frac{\Longrightarrow \langle \pi\ \ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \ l_1\texttt{:}\ldots l_k\texttt{:\{ break;}\ p\ \texttt{\}}\ \omega\rangle\phi}$$

$$\text{blockBreakLabel}\ \frac{\Longrightarrow \langle \pi\ \ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \ l_1\texttt{:}\ldots l_i\texttt{:}\ldots l_k\texttt{:\{ break}\ l_i\texttt{;}\ p\ \texttt{\}}\ \omega\rangle\phi}$$

To blocks (labeled or unlabeled) that are abruptly terminated by a `break` statement with a label l not matching any of the labels of the block, the following rule applies:

$$\text{blockBreakNomatch}\ \frac{\Longrightarrow \langle \pi\ \texttt{break}\ l;\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ l_1\texttt{:}\ldots l_k\texttt{:\{}\ \texttt{break}\ l;\ p\texttt{\}}\ \omega\rangle\phi}$$

Similar rules exist for blocks that are terminated by a `return` or `throw` statement:

$$\text{blockReturn}\ \frac{\Longrightarrow \langle \pi\ \texttt{return}\ v;\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ l_1\texttt{:}\ldots l_k\texttt{:\{}\ \texttt{return}\ v;\ p\texttt{\}}\ \omega\rangle\phi}$$

$$\text{blockThrow}\ \frac{\Longrightarrow \langle \pi\ \texttt{throw}\ v;\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ l_1\texttt{:}\ldots l_k\texttt{:\{}\ \texttt{throw}\ v;\ p\texttt{\}}\ \omega\rangle\phi}$$

3.7 Abstraction and Modularization Rules

The symbolic execution rules presented so far are sufficient to verify many safety properties of Java programs. With these rules, method declarations are inlined at the invocation site and loops are unwound. Verifying programs this way is very similar to using a bounded model checker, such as, for example, CBMC [Kroening and Tautschnig, 2014].

Yet, in order for program verification to scale up, *abstraction* is in general required. With abstraction, certain pieces of code being verified are replaced with an approximation. The term "abstraction" refers to both the process and the approximation used.

Before we give a definition, let's recall that every program fragment p induces a transition relation $\rho(p)$ on states (Definition 3.5).

Definition 3.21 (Abstraction). We call a relation $\alpha(p)$ on states an *abstraction* of p, iff

$$\rho(p) \subseteq \alpha(p) \ , \tag{3.1}$$

i.e., iff the abstraction $\alpha(p)$ contains all behaviors that the program p exhibits (or more).

The two major kinds of abstractions in KeY are method contracts and loop invariants. They are user-supplied but machine-checked for correctness. The user describes an abstraction syntactically using JavaDL or, more often, JML. KeY generates a proof obligation that the abstraction is correct, i.e., that it fulfills (3.1). In parallel, the abstraction can be used in place of the abstracted method or loop.

Abstraction offers several advantages:

1. Not all aspects of the code are crucial to establish a given correctness property. Abstraction allows eliding irrelevant aspects, thus reducing proof size and complexity.

2. Abstractions can be used to facilitate inductive reasoning, such as is the case with loop invariants.
3. Abstractions can be checked once and used multiple times, potentially saving proof effort.
4. When a part of the program is extended or modified, it is sufficient to check that the new version conforms to the same abstraction as the old one. It is not necessary to reverify the rest of the program.
5. For certain program parts (library code, native code) the source code may be unavailable. A user-provided abstraction is a convenient way to capture some or all of the missing code's functionality.

Advantages 3 and 4 are typically what is referred to as *modularization*.

At the same time, there are also costs to using abstraction. One of them is associated overhead. For simple methods, it might be more efficient to inline the method implementation instead of writing, proving, and using a contract. Another one is incompleteness. If a proof attempt cannot be completed, an insufficiently precise abstraction can be the reason. The user needs to diagnose the issue and refine the abstraction.

In the following, we briefly introduce the method contract and the loop invariant rules of JavaDL.

3.7.1 Replacing Method Calls by Specifications: Method Contracts

The specification of a method is called *method contract* and is defined as follows (this definition is identical to Definition 8.2 on page 268, where the translation of JML contracts into JavaDL is presented).

Definition 3.22 (Functional method contract). A functional JavaDL method contract for a method or constructor

$$R\ \mathtt{m}(T_1\ \mathtt{p}_1,\ \ldots,\ T_n\ \mathtt{p}_n)$$

declared in class C is a quadruple

$$(pre, post, mod, term)$$

that consists of

- a precondition $pre \in \mathrm{DLFml}$,
- a postcondition $post \in \mathrm{DLFml}$,
- a modifier set $mod \in \mathrm{Trm}_{LocSet} \cup \{\textsc{strictlyNothing}\}$, and
- a termination witness $term \in \mathrm{Trm}_{Any} \cup \{\textsc{partial}\}$.

Contract components may contain special program variables referring to the execution context:

- $\mathtt{self} : C$ for references to the receiver object (not available if `m` is a static method),

- $\mathtt{p}_1 : T_1, \ldots, \mathtt{p}_n : T_n$ representing the method's formal parameters,
- $\mathtt{heap} : \mathit{Heap}$ to access heap locations,
- $\mathtt{heap}^{pre} : \mathit{Heap}$ to access heap locations in the state in which the operation was invoked (in the postcondition only),
- $\mathtt{exc} : \mathit{Exception}$ to refer to the exception in case the method terminates abruptly with a thrown exception (in the postcondition only),
- $\mathtt{res} : R$ to refer to the result value of a method with a return type different from `void` (in the postcondition only).

While $pre, mod, term$ (only) refer to the state before method invocation, the postcondition *post* refers (also) to the state after termination of the invoked method. Therefore, *post* has more expressive means to its avail: Besides two heap representations (`heap` and $\mathtt{heap}^{pre}$), the result value, and a possibly thrown exception can be used in the postcondition. In some situations, certain context variables are not available. For instance, there is no result value for a constructor invocation.

Usually (especially when employing JML as specification language), the postcondition $post \in \mathrm{DLFml}$ is of the form

$$(\mathtt{exc} \doteq \mathtt{null} \rightarrow \phi) \land (\mathtt{exc} \not\doteq \mathtt{null} \rightarrow \psi) \ ,$$

where ϕ is the postcondition for the case that the method terminates normally and ψ is the postcondition in case the method terminates abruptly with an exception.

The formulas *pre* and *post* are JavaDL formulas. However, in most cases, they do not contain modal operators. This is in particular true if they are automatically generated translations of JML specifications.

The termination marker *term* can be the special value PARTIAL, indicating that the contract is partial and does not require the method to terminate. Alternatively, *term* is an expression whose value needs to be decreasing according to some well-founded ordering with every recursive call. If the method does not involve recursive calls, any expression can be used for *term* (e.g., zero). More on termination proofs for recursive methods can be found in Section 9.1.4.

Below, we give the rule methodContractPartial that replaces a method invocation during symbolic execution with the method's contract. The rule assumes that the given method contract is a correct abstraction of the method. There must be a separate argument (i.e., a separate proof) establishing this fact. Chapter 8 gives details on such correctness arguments for method contracts.

The rule methodContractPartial applies to a box-modality and, thus, the question of whether the method terminates is ignored.

The above rule is applicable to a method invocation in which the receiver se_{target} and the arguments se_i are simple expressions. This can be achieved by using the rules methodCallUnfoldTarget and methodCallUnfoldArguments introduced in Section 3.6.5.3.

In the first premiss, we have to show that the precondition *pre* holds in the state in which the method is invoked after updating the program variables `self` and $\mathtt{p}_i$ with the receiver object se_{target} and with the parameters se_i. This guarantees that the methodContractPartial

$$\frac{\begin{array}{l}\Longrightarrow \mathscr{U}^{pre}_{cont}\, pre \\ \mathscr{U}^{post}_{cont}\mathscr{A}_{mod}(\texttt{exc} \doteq \texttt{null}) \Longrightarrow \mathscr{U}^{post}_{cont}\mathscr{A}_{mod}(post \rightarrow \{lhs := \texttt{res}\}\ [\pi\ \omega]\phi) \\ \mathscr{U}^{post}_{cont}\mathscr{A}_{mod}(\texttt{exc} \not\doteq \texttt{null}) \Longrightarrow \mathscr{U}^{post}_{cont}\mathscr{A}_{mod}(post \rightarrow [\pi\ \texttt{throw}\ exc\texttt{;}\ \ \omega]\phi)\end{array}}{\Longrightarrow [\pi\ \ lhs\texttt{=}se_{target}\texttt{.}method\texttt{(}se_1,\ldots,se_n\texttt{);}\ \ \omega]\phi}$$

where

- $(pre, post, mod, term)$ is a contract for *method*;
- $\mathscr{U}^{pre}_{cont} = \{\texttt{self} := se_{target}\, ||\, p_1 := se_1\, ||\, \cdots\, ||\, p_n := se_n\}$ is an update application setting up the precondition-context variables;
- $\mathscr{U}^{post}_{cont} = \mathscr{U}^{pre}_{cont}\{\texttt{heap}^{pre} := \texttt{heap}\, ||\, \texttt{res} := c_r\, ||\, \texttt{exc} := c_e\}$ is an update application setting up the postcondition-context variables; c_r and c_e are fresh constants of the result type of *method* or of type `Throwable`;
- $\mathscr{A}_{mod}$ is an anonymizing update w.r.t. the modifier set *mod*.

Figure 3.7 Method contract rule

method contract's precondition is fulfilled and, according to the contract, we can use the postcondition *post* to describe the effects of the method invocation—where two cases must be distinguished.

In the first case (second premiss), we assume that the invoked method terminates normally, i.e., the context variable `exc` is `null` after termination. If the method is nonvoid the return value `res` is assigned to the variable *lhs*. The second case deals with the situation that the method terminates abruptly (third premiss). As in the normal-termination case, the context variables are updated with the corresponding terms. But now, there is no result value to be assigned, but the exception `exc` is thrown explicitly.

Note that, in both cases, the locations that the method possibly modifies are updated with an anonymizing update $\mathscr{A}_{mod}$. Such an update, which replaces the values of the locations in *mod* with new anonymous values can be constructed using the function $anon : Heap \times LocSet \times Heap \rightarrow Heap$ (see Section 2.4.3). The heap update

$$\{\texttt{heap} := anon(\texttt{heap}, mod, h)\}\ ,$$

where h is a new constant of type *Heap*, ensures that, in its scope, the heap coincides with h on all locations in *mod* and all not yet created locations and coincides with `heap` before the update elsewhere.

Anonymizing the locations in *mod* ensures that the only knowledge that can be used about these locations when reasoning about the poststate is the knowledge contained in *post*—and not any knowledge that may be contained in other formulas in the sequence, which in fact refers to the prestate. Otherwise, without anonymization, knowledge about the pre- and the poststate would be mixed in an unsound way. See Section 9.4.1 for further information on the concept of anonymizing updates.

The method contract rule for the box modality is similar. It can be applied independently of the value of the termination marker.

3.7.2 Reasoning about Unbounded Loops: Loop Invariants

Loops with a small bound on the number of iterations can be handled by loop unwinding (see Section 3.6.4). If, however, there is no bound that is known a priori or if that bound is too high, then unwinding does not work. In that case, a loop-invariant rule has to be used.

A loop invariant is a formula describing an overapproximation of all states reachable by repeated execution of a loop's body while the loop condition is true. Using a loop invariant essentially is an inductive argument, proving that the invariant holds for any number of loop iterations—and, thus, still holds when the loop terminates.

Loop invariant rules are probably the most involved and complex rules of the KeY system's JavaDL calculus. This complexity results from the inductive structure of the argument but also from the features of Java loops, which include the possibility of side effects and abrupt termination in loop conditions and loop bodies.

In this section, we present basic versions of the loop invariant rules; in particular, loop termination and using the loop's modifier set for framing is not considered in the following. Enhanced loop invariant rules are presented in Chapter 9. Moreover, Section 16.3 provides a more intuitive introduction to formal verification of while loops with invariants and contains a tutorial on systematic development of loop invariants.

Also, automatic invariant generation is a hot research topic—a particular approach to this challenge is described in Section 6.3.

The loop invariant rule has been cross-verified against another language framework for an earlier version of JavaDL [Widmann, 2006].

3.7.2.1 Loop Specifications

A loop specification is similar to a method contract in that it formalizes an abstraction of the relationship between the state before a method or loop is executed and the state when the method or loop body, respectively, terminates. In that sense, a loop invariant is both the pre- and the postcondition of the loop body. Yet, in most cases, a useful loop invariant is more difficult to find than a method contract because it relates the initial state with the states after *every* loop iteration.

Like method contracts, loop specifications contain two additional elements: (a) a modifier set describing which parts of the state the loop body can modify and (b) a termination witness providing an argument for the loop's termination. As said above, the basic rules presented in this chapter do not make use of this additional information. Extended rules considering modifier sets and termination are presented in Chapter 9.

Definition 3.23. A *loop specification* is a tuple

$$(inv, mod, term)$$

that consists of

- a loop invariant $inv \in$ DLFml,
- a modifier set $mod \in \mathrm{Trm}_{LocSet} \cup \{\textsc{strictlyNothing}\}$.
- a termination witness $term \in \mathrm{Trm}_{Any} \cup \{\textsc{partial}\}$.

Specification components may make use of special program variables which allow them to refer to the execution context:

- all local variables that are defined in the context of the loop,
- `self` : C for references to the receiver object of the current method frame (not available if that frame belongs to a static method),
- `heap` : *Heap* referring to the heap in the state after the current iteration,
- $\mathtt{heap}^{pre}$: *Heap* referring to the heap in the initial state of the immediately enclosing method frame.

3.7.2.2 Basic Version of the Loop Invariant Rule

The first basic loop invariant rule we consider makes two assumptions: It is only applicable if (1) the loop guard is a simple expression *se*, i.e., the loop condition cannot have side effects and cannot terminate abruptly. And (2) the loop body p_{norm} must be guaranteed to always terminate normally, i.e.,

1. execution of p_{norm} does not raise an exception, and
2. p_{norm} does not contain `break`, `continue`, `return` statements.

The rule takes the form shown in Figure 3.8.

$$\mathsf{simpleInv}\ \frac{\begin{array}{l}\Longrightarrow inv\\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((inv \land se \doteq TRUE) \ \rightarrow\ [p_{norm}]inv\big)\\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((inv \land se \doteq FALSE) \ \rightarrow\ [\pi\ \omega]\varphi\big)\end{array}}{\Longrightarrow [\pi\ \mathtt{while}(se)\ \mathtt{\{}\ p_{norm}\ \mathtt{\}}\ \omega]\varphi}$$

where
- *se* is a simple expression and p_{norm} cannot terminate abruptly;
- $(inv, mod, term)$ is a loop specification for the loop to which the rule is applied;
- $\mathscr{A}_{heap} = \{\mathtt{heap} := c_h\}$ anonymizes the heap; c_h:*Heap* is a fresh constant;
- $\mathscr{A}_{local} = \{l_1 := c_1 \parallel \cdots \parallel l_n := c_n\}$ anonymizes all local variables $l_1, \ldots, l_n$ that are the target of an assignment (left-hand side of an assignment statement) in p_{norm}; each c_i is a fresh constant of the same type as l_i.

Figure 3.8 Basic loop invariant rule

When a method contract is used for verification, the validity of the contract is not part of the premisses of the contract rule but a separate proof obligation. In contrast to that, the loop invariant rule combines both aspects in its three premisses.

- *Base case:* The first premiss is the base case of the inductive argument. One has to show that the invariant is satisfied whenever the loop is reached.

- *Step case:* The second premiss is the inductive step. One has to show that, if the invariant holds before execution of the loop body, then it still holds afterwards.[6]
- *Use case:* The third premiss uses the inductive argument and continues the symbolic execution for the code $\pi\ \omega$ following the loop but now with the knowledge that the invariant holds.

Note that, in the step case, one can assume the loop condition *se* to be *TRUE* (i.e., the loop is iterated once more). In the use case, on the other hand, one can assume that the loop condition *se* is *FALSE* (i.e., the loop has terminated).

The combination $\mathscr{A}_{heap}\mathscr{A}_{local}$ is called the *anonymizing update application* of the loop rule. It needs to be added to the second and the third premiss of the rule, which refer to the state after an unknown number of loop iterations. Its application ensures that only the knowledge encoded in *inv* can be used to reason about the heap locations and the local variables changed by the loop. Instead, of using the update application $\mathscr{A}_{heap}$ that anonymizes all heap locations, one can use a more precise update $\mathscr{A}_{mod}$ that only anonymizes the locations in *mod* (see the previous section on method contracts and Section 9.4.1 for more information). This requires, however, the additional proof that the loop body does indeed not modify any other locations than those in *mod*.

3.7.2.3 Loop Conditions with Side-effects

In Java, loop conditions may have side effects. For example, the loop condition in

```
while(a[i++] > 0) { ... }
```

has a side effect on the local variable `i`.

In Figure 3.9, we present a loop invariant rule that allows the loop condition to be a nonsimple expression *nse*, i.e., to have side effects. The idea is to capture the value of *nse* in a fresh Boolean program variable `b`. To account for the effects of the condition, its evaluation is repeated right before the loop body.

While the rule sideEffectInv takes into account any state changing side effects in *nse*, it does not yet capture the exceptions that it might throw. For example, the possibility that the loop condition `a[i++]` in the above example can throw an `ArrayIndexOutOfBoundsException` is not considered.

3.7.2.4 Loops with Abrupt Termination

In the loop invariant rules shown above (simpleInv and sideEffectInv), the loop body is executed outside its usual context $\pi\ \omega$. Thus, a `continue` or a `break` statement does not make sense. Likewise, a `return` statement is not sensible since it is not

[6] Note that the loop body in the step case is not enclosed in the execution context $\pi\ \omega$. Nevertheless, the innermost method frame that is part of π has to be added implicitly so that method invocations within p_{norm} can be resolved correctly.

$$\mathsf{sideEffectInv}\ \frac{\begin{array}{l}\Longrightarrow inv \\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((inv \land [\texttt{b=}nse\texttt{;}]\texttt{b} \doteq TRUE) \rightarrow [\texttt{b=}nse\texttt{;}\ p_{norm}]inv\big) \\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((inv \land [\texttt{b=}nse\texttt{;}]\texttt{b} \doteq FALSE) \rightarrow [\pi\ \texttt{b=}nse\texttt{;}\ \omega]\varphi\big)\end{array}}{\Longrightarrow [\pi\ \texttt{while(}nse\texttt{)}\ \texttt{\{}\ p_{norm}\ \texttt{\}}\ \omega]\varphi}$$

where

- p_{norm} and nse cannot terminate abruptly;
- $(inv, mod, term)$ is a loop specification for the loop to which the rule is applied;
- $\mathscr{A}_{heap} = \{\texttt{heap} := c_h\}$ anonymizes the heap; c_h:*Heap* is a fresh constant;
- $\mathscr{A}_{local} = \{l_1 := c_1 \parallel \cdots \parallel l_n := c_n\}$ anonymizes all local variables $l_1, \ldots, l_n$ that are the target of an assignment (left-hand side of an assignment statement) in p_{norm} or in nse; each c_i is a fresh constant of the same type as l_i;
- b is a fresh Boolean variable.

Figure 3.9 Invariant rule for loops with side effects in the loop condition

embedded into the original method frames, and exceptions do not occur within the right `try-catch-finally` block.

In order to be able to deal with loop bodies in isolation, we transform them in such a way that abnormal termination is turned into normal termination in which certain flags are set signaling the abnormal termination. We will not go into details of this transformation here, but illustrate it using one synthetic example loop, which exhibits all possible reasons for abnormal termination:

```
while(x >= 0) {
  if(x == 0) break;
  if(x == 1) return 42;
  if(x == 2) continue;
  if(x == 3) throw e;
  if(x == 4) x = -1;
}
```

We use the Boolean variables BREAK and RETURN, and a variable EXCEPTION of type `Throwable` to store and signal the termination state of the loop body. In the example, the original loop body is translated into the block

```
loopBody: {
  try {
    BREAK=false; RETURN=false; EXCEPTION=null;
    if(x == 0) { BREAK=true; break loopBody; }
    if(x == 1) { res=42; RETURN=true; break loopBody; }
    if(x == 2) { break loopBody; }
    if(x == 3) { throw e; }
    if(x == 4) { x = -1; }
  } catch(Throwable e) {
    EXCEPTION = e;
  }
}
```

The result of the transformation is guaranteed to terminate normally, with the original termination reason caught in the Boolean flags.

In general, this transformation can be more involved if it has to deal with nested labeled blocks and loops. It then resembles the translation outlined in Section 3.6.4 for loop unwinding.

Using the above transformation, the loop invariant rules that can handle both abrupt termination and side effects in the loop condition takes the form shown in Figure 3.10.

In the second premiss of this rule (subformula *post*), if a loop is left via abnormal termination rather than by falsifying the loop condition, the loop invariant does not need be reestablished but the execution of the program in its original context $\pi\ \omega$ is resumed—retriggering an exception or return statement if they were observed in the loop body. The rationale behind this is that loop invariants are supposed to hold whenever the loop is potentially reentered, which is not the case if a `return`, `throw`, or `break` statement has been executed. If, however, a `continue` statement is executed in the loop body p, the transformation $\widehat{\mathtt{b}{=}nse;\, p}$ terminates normally and the invariant has to hold before the next loop iteration is started (as in the NORMAL case).

In this chapter, we have not presented a loop invariant rule that handles loops in $\langle\cdot\rangle$-modalities and, thus, needs to guarantee program termination; this issue is addressed in Section 9.4.2. One aspect shall be mentioned here nonetheless: When termination matters, the modality $\langle\cdot\rangle$ is used instead of $[\cdot]$. However, the box modality $[\mathtt{b}{=}nse]$, which occurs on the left-hand side of the second and the third premiss in rules sideEffectInv (Figure 3.9) and abruptTermInv (Figure 3.10), must remain a box modality. If it were to be changed into a diamond modality, then a nonterminating loop condition would make these two premisses of the loop invariant rules trivially valid; the calculus would be unsound.

$$\mathsf{abruptTermInv}\ \frac{\begin{array}{l}\Longrightarrow \mathit{inv}\\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((\mathit{inv} \wedge [\mathtt{b}\texttt{=}\mathit{nse};]\mathtt{b} \doteq \mathit{TRUE}) \ \rightarrow\ [\widehat{\mathtt{b}\texttt{=}\mathit{nse};\,p}]\mathit{post}\big)\\ \Longrightarrow \mathscr{A}_{heap}\mathscr{A}_{local}\big((\mathit{inv} \wedge [\mathtt{b}\texttt{=}\mathit{nse};]\mathtt{b} \doteq \mathit{FALSE}) \ \rightarrow\ [\pi\ \mathtt{b}\texttt{=}\mathit{nse};\ \omega]\varphi\big)\end{array}}{\Longrightarrow [\pi\ \mathtt{while(}\mathit{nse}\mathtt{)\ \{}\ p\ \mathtt{\}}\ \omega]\varphi}$$

where

- $(\mathit{inv}, \mathit{mod}, \mathit{term})$ is a loop specification for the loop to which the rule is applied;
- $\mathscr{A}_{heap} = \{\mathtt{heap} := c_h\}$ anonymizes the heap; $c_h{:}\mathit{Heap}$ is a fresh constant;
- $\mathscr{A}_{local} = \{l_1 := c_1 \parallel \cdots \parallel l_n := c_n\}$ anonymizes all local variables $l_1, \ldots, l_n$ that are the target of an assignment (left-hand side of an assignment statement) in p_{norm} or in *nse*; each c_i is a fresh constant of the same type as l_i;
- `b` is a fresh Boolean variable;
- $\widehat{\mathtt{b}\texttt{=}\mathit{nse};\,p}$ is the result of transforming $\mathtt{b}\texttt{=}\mathit{nse};\,p$ as described above to handle abrupt termination;
- *post* is the formula

$$\begin{array}{lll} & (\textsc{Exception} \not\doteq \mathit{null} \rightarrow & [\pi\ \mathtt{throw\ \textsc{Exception};}\ \omega]\varphi)\\ \wedge & (\textsc{Break} \doteq \mathit{TRUE} \rightarrow & [\pi\ \omega]\varphi)\\ \wedge & (\textsc{Return} \doteq \mathit{TRUE} \rightarrow & [\pi\ \mathtt{return\ res;}\ \omega]\varphi)\\ \wedge & (\textsc{Normal} \rightarrow & \mathit{inv}) \end{array}$$

with

$$\begin{array}{rcl}\textsc{Normal} & \equiv & \textsc{Break} \doteq \mathit{FALSE}\ \wedge\\ & & \textsc{Return} \doteq \mathit{FALSE}\ \wedge\\ & & \textsc{Exception} \doteq \mathit{null}\end{array}$$

Figure 3.10 Invariant rule for loops with abrupt termination