3

Dynamic Logic

by

Bernhard Beckert
Vladimir Klebanov
Steffen Schlager

3.1 Introduction

In the previous chapter, we have introduced a variant of classical predicate logic that has a rich type system and a sequent calculus for that logic. This predicate logic can easily be used to describe and reason about data structures, the relations between objects, the values of variables—in short: about the states of (JAVA) programs.

Now, we extend the logic and the calculus such that we can describe and reason about the behaviour 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] as the logical basis of the KeY system's software verification component [Beckert, 2001]. The principle of DL is the formulation of statements about program behaviour by integrating programs and formulae within a single language. To this end, the operators (modalities) $\langle p \rangle$ and $[p]$ can be used in formulae, where p can be any sequence of legal JAVA CARD statements (i.e., DL is a multi-modal logic). These operators that refer to the final state of p 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 $\texttt{x} \doteq 0 \rightarrow \langle \texttt{x++} \rangle (\texttt{x} \doteq 1)$.

In general, there can be more than one final state because the programs can be non-deterministic; but here, since JAVA CARD programs are deterministic, there is exactly one such state (if p terminates normally, i.e., does not terminate abruptly due to an uncaught exception) or there is no such state (if p does not terminate or terminates abruptly). "Deterministic" here means that a program, for the same initial state and the some inputs, always has the

same behaviour—in particular, the same final state (if it terminates) and the same outputs. When we do not (exactly) know what the initial state resp. the inputs are, we may not know what (exactly) the behaviour is. But that does not contradict determinism of the programming language JAVA CARD.

Deduction in DL, and in particular in JAVA CARD DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of JAVA (⇒ Sect. 3.4.5).

Dynamic Logic and Hoare Logic

Dynamic logic can be seen as an extension of Hoare logic. The DL formula $\phi \texttt{->} [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulae of DL is closed under the usual logical operators: In Hoare logic, the formulae ϕ and ψ are pure first-order formulae, whereas in DL they can contain programs.

DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Also, all JAVA constructs are available in our DL for the description of states (including `while` loops and recursion). It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type; instead DL formulae can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

Structure of this Chapter

The structure of this chapter is similar to that of the chapter on first-order logic. We first define syntax and semantics of our JAVA CARD dynamic logic in Sections 3.2 and 3.3. Then, in Section 3.4–3.9, we present the JAVA CARD DL calculus, which is used in the KeY system for verifying JAVA CARD programs. Section 3.4 gives an overview, while Sect. 3.5–3.9 describe the main components of the calculus: non-program rules (Sect. 3.5), rules for reducing JAVA CARD programs to combinations of state updates and case distinctions (Sect. 3.6), rules for handling loops with the help of loop invariants (Sect. 3.7), rules for handling method calls with the help of method contracts (Sect. 3.8), and the simplification and normalisation of state updates (Sect. 3.9). Finally, Sect. 3.10 discusses related work.

In addition, some important aspects of JAVA CARD DL and the calculus are discussed in other chapters of this book, including the first-order part (Chapter 2), proof construction and search (Chapter 4), induction (Chapter 11), handling integers (Chapter 12), and handling the particularities of JAVA CARD such as JAVA CARD's transaction mechanism (Chapter 9). An introduction to using the implementation of the calculus in the KeY system is given in Chapter 10.

3.2 Syntax

In general, a dynamic logic is constructed by extending some non-dynamic logic with parameterised modal operators $\langle p \rangle$ and $[p]$ for every legal program p of some programming language.

In our case, the non-dynamic base logic is the typed first-order predicate logic described in Chapter 2. Not surprisingly, the programming language we consider is JAVA CARD, i.e., the programs p within the modal operators are in our case JAVA CARD programs. The logic we define in this section is called JAVA CARD Dynamic Logic or, for short, JAVA CARD DL.

The structure of this section follows the structure of Chapter 2. Sect. 3.2.1 defines the notions of type hierarchy and signature for JAVA CARD DL. However, we are more restrictive here than in the corresponding definitions of Chapter 2 (Def. 2.1 and 2.8) since we want the JAVA CARD DL type hierarchy to reflect the type hierarchy of JAVA CARD. A JAVA CARD DL type hierarchy must, e.g., always contain a type Object. Then, we define the syntax of JAVA CARD DL which consists of terms, formulae, and a new category of expressions called *updates* (Sect. 3.2). In the subsequent Sect. 3.3, we present a model-theoretic semantics of JAVA CARD DL based on Kripke structures.

3.2.1 Type Hierarchy and Signature

We start with the definition of the underlying type hierarchies and the signatures of JAVA CARD DL. Since the logic we define is tailored to the programming language JAVA CARD, we are only interested in type hierarchies containing a set of certain types that are part of every JAVA CARD program. First, we define a direct subtype relation that is needed in the subsequent definition.

Definition 3.1 (Direct subtype). *Assume a first-order logic type system* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$. *Then the* direct subtype relation $\sqsubseteq^0 \subseteq \mathcal{T} \times \mathcal{T}$ *between two types* $A, B \in \mathcal{T}$ *is defined as:*

$$\begin{gathered} A \sqsubseteq^0 B \\ \textit{iff} \\ A \sqsubseteq B \textit{ and } A \neq B \textit{ and} \\ C = A \textit{ or } C = B \textit{ for any } C \in \mathcal{T} \textit{ with } A \sqsubseteq C \textit{ and } C \sqsubseteq B. \end{gathered}$$

Intuitively, A is a direct subtype of B if there is no type C that is between A and B.

Definition 3.2 (JAVA CARD DL type hierarchy). *A* JAVA CARD DL type hierarchy *is a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *(⇒ Def. 2.1) such that:*

- $\mathcal{T}_d$ *contains (at least) the types:*

`integerDomain`, boolean, Object, Error, Exception, RuntimeException, NullPointerException, ClassCastException, ExceptionInInitializerError, ArrayIndexOutOfBoundsException, ArrayStoreException, ArithmeticException, Null;

- $\mathcal{T}_a$ *contains (at least) the types:* `integer`, `byte`, `short`, `int`, `long`, `char`, Serializable, Cloneable, Throwable;
- *if* $A \sqsubseteq$ Object, *then* Null $\sqsubseteq A$ *for all* $A \neq \bot \in \mathcal{T}$;
- `integerDomain` $\sqsubseteq^0 A$ *and* $A \sqsubseteq^0$ `integer` *for all* $A \in \{$`byte`, `short`, `int`, `long`, `char`$\}$;
- $\bot \sqsubseteq^0$ `integerDomain`;
- $\bot \sqsubseteq^0$ Null;
- $\bot \sqsubseteq^0$ boolean;
- $A \sqcap B = \bot$ *for all* $A \in \{$`integerDomain`, `integer`, `byte`, `short`, `int`, `long`, `char`, boolean$\}$ *and* $B \sqsubseteq$ Object.

In the remainder of this chapter, with type hierarchy *we always mean a* Java Card DL type hierarchy, *unless stated otherwise.*

A Java Card DL type hierarchy is a type hierarchy containing the types that are built into Java Card like boolean, the root reference type Object, and the type Null, which is a subtype of all reference types (Null exists implicitly in Java Card). As in the first-order case, the type hierarchy contains the special types $\top$ and $\bot$ (⇒ Def. 2.1). Moreover, it contains a set of abstract and dynamic (i.e., non-abstract) types reflecting the set of Java Card interfaces and classes necessary when dealing with arrays. These are Cloneable, Serializable, Throwable, and some particular sub-sorts of the latter which are the possible exceptions and errors that may occur during initialisation and when working with arrays.

Finally, a type hierarchy includes the types boolean, `byte`, `short`, `int`, `long`, and `char` representing the corresponding primitive Java Card types. Note, that these types (except for boolean) are abstract and are subtypes of the likewise abstract type `integer`. The common subtype of these types is the non-abstract type `integerDomain`, thus satisfying the requirement that any abstract type must have a non-abstract subtype. Later we define that the domain of `integerDomain` is the (infinite) set $\mathbb{Z}$ of integer numbers. Since the domain of a type by definition (⇒ Def. 2.20) includes the domains of its subtypes, all the abstract supertypes of `integerDomain` share the common domain $\mathbb{Z}$. The typing of the usual functions on the integers, like e.g., addition, is defined as `integer`, `integer` → `integer`.

Reasons for the Complicated Integer Type Hierarchy

The reasons behind the somewhat complicated looking integer type hierarchy are twofold. First, we want to have mathematical integers in the logic instead of integers with finite range as in Java Card (the advantage of this decision is explained in Chapter 12). As a consequence, the

type `integer` is defined. Second, we need a dedicated type for each primitive JAVA CARD integer type. That is necessary for a correct handling of `ArrayStoreException`s in the calculus which requires the mapping between types in JAVA CARD and sorts in JAVA CARD DL to be an injection.

The reason why we introduce the common subtype `integerDomain` is that `integer`, `short`, ..., `char` are supposed to share the same domain, namely the integer numbers $\mathbb{Z}$. As a consequence of Def. 2.20, which requires that any domain element $d \in \mathcal{D}$ has a unique dynamic type $\delta(d)$, the only possibility to obtain the same domain for several types is to declare these types as abstract and introduce a common non-abstract subtype holding the domain.

The definition of type hierarchies (Def. 3.2) partly fixes the subtype relation. It requires that type Null is a common subtype of all subtypes of Object (except $\bot$). That is necessary to correctly reflect the JAVA CARD reference type hierarchy. Besides reference types, JAVA CARD has primitive types (e.g., `boolean`, `byte`, or `int`) which have no common sub- or supertype with any reference type. Def. 3.2 guarantees that we only consider type hierarchies where there is no common subtype (except $\bot$, which does not exist in JAVA CARD) of primitive and reference types, thus correctly reflecting the type hierarchy of the JAVA CARD language.

However, Def. 3.2 does not fix the set of user-defined types and the subtype relation between them. A JAVA CARD type hierarchy can contain additional user-defined types, e.g., those types that are declared in a concrete JAVA CARD program (⇒ Def. 3.10).

Fig. 3.1 shows the basic type hierarchy without any user-defined types. Due to space restrictions the types `short`, `int`, and the built-in API reference types like Serializable, Cloneable, Exception, etc. are omitted from the figure. Abstract types are written in italics ($\bot$ is of course also abstract). The subtype relation $A \sqsubseteq B$ is illustrated by an arrow from A to B (reflexive arrows are omitted).

Example 3.3. Consider the type hierarchy in Fig. 3.2 which is an extension of the first-order type hierarchy from Example 2.6. The types AbstractCollection, List, AbstractList, ArrayList, and the array type Object[] are user-defined, i.e., are not required to be contained in any type hierarchy. As we define later, any array type must be a subtype of the built-in types Object, Serializable, and Cloneable.

Due to space restrictions some of the built-in types (⇒ Fig. 3.1) are omitted.

As in Chapter 2, we now define the set of symbols that the language JAVA CARD DL consists of. In contrast to first-order signatures, we have two kinds of function and predicate symbols: *rigid* and *non-rigid* symbols. Consequently, the set of function symbols is divided into two disjoint subsets FSym_r

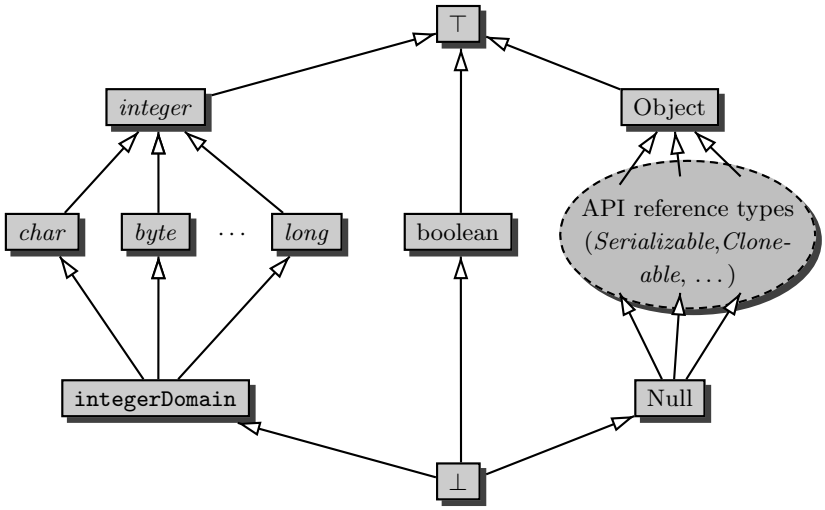


Fig. 3.1. Basic Java Card DL type hierarchy without user-defined types

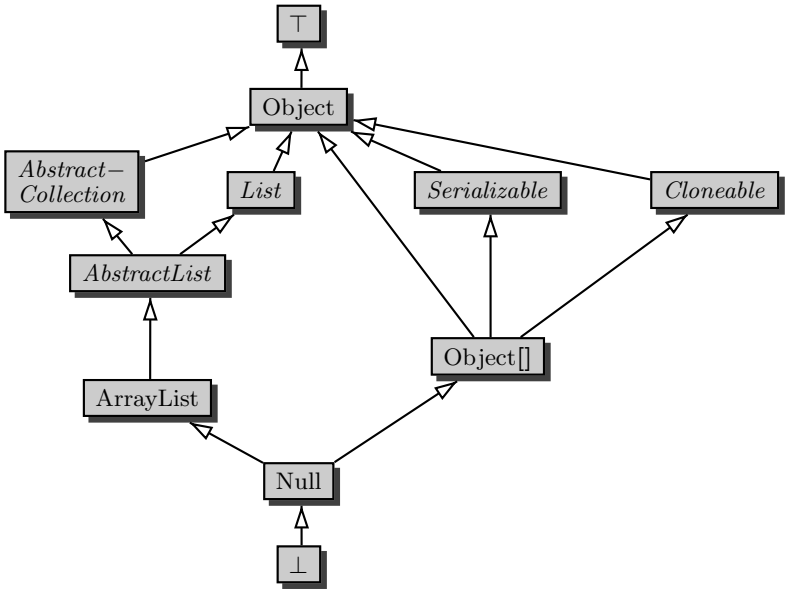


Fig. 3.2. Example for a Java Card DL type hierarchy (built-in types partly omitted)

and FSym_{nr} of rigid and non-rigid functions, respectively (the same applies to the set of predicate symbols). Intuitively, rigid symbols have the same meaning in all program states (e.g., the addition on integers or the equality predicate), whereas the meaning of non-rigid symbols may differ from state to state. Non-rigid symbols are used to model (local) variables, attributes, and arrays outside of modalities, i.e., they occur as terms in JAVA CARD DL. Local variables can thus not be bound by quantifiers—in contrast to logical variables. Note, that in classical DL there is no distinction between logical variables and program variables (constants).

We only allow signatures that contain certain function and predicate symbols. For example, we require that a JAVA CARD DL signature contains constants $0, 1, \ldots$ representing the integer numbers, function symbols for arithmetical operations like addition, subtraction, etc., and the typical ordering predicates on the integers.

Definition 3.4 (JAVA CARD DL signature). *Let T be a type hierarchy, and let FSym^0_r , FSym^0_{nr}, PSym^0_r , and PSym^0_{nr} be the sets of rigid and non-rigid function and predicate symbols from App. A.*

Then, a JAVA CARD DL signature *(for T) is a tuple*

$$\Sigma = (\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$$

consisting of

- *a set* VSym *of variables (as in the first-order case, Def. 2.8),*
- *a set* FSym_r *of* rigid *function symbols and a set* FSym_{nr} *of* non-rigid *function symbols such that*

$$\begin{aligned} &\mathrm{FSym}_r \cap \mathrm{FSym}_{nr} = \emptyset \\ &\mathrm{FSym}^0_r \subseteq \mathrm{FSym}_r \\ &\mathrm{FSym}^0_{nr} \subseteq \mathrm{FSym}_{nr}, \end{aligned}$$

- *a set* PSym_r *of* rigid *predicate symbols and a set* PSym_{nr} *of* non-rigid *predicate symbols such that*

$$\begin{aligned} &\mathrm{PSym}_r \cap \mathrm{PSym}_{nr} = \emptyset \\ &\mathrm{PSym}^0_r \subseteq \mathrm{PSym}_r \\ &\mathrm{PSym}^0_{nr} \subseteq \mathrm{PSym}_{nr}, \textit{ and} \end{aligned}$$

- *a typing function α (as in the first-order case, Def. 2.8).*

In the remainder of this chapter, with signature *we always mean a* JAVA CARD DL signature*, unless stated otherwise.*

Example 3.5. Given the type hierarchy from Example 3.3 (⇒ Fig. 3.2), an example for a signature is the following:

$$\mathrm{VSym} = \{a, n, x\}$$

with

$$a{:}\text{ArrayList}, \quad n{:}\texttt{integer}, \quad x{:}\texttt{integer}$$

$$\mathrm{FSym}_r = \{f, g\} \cup \mathrm{FSym}_r^0$$

with

$$f : \texttt{integer} \to \texttt{integer}$$
$$g : \texttt{integer}$$

$$\mathrm{FSym}_{nr} = \{al, arg, c, data, i, j, length, para1, sal, v\} \cup \mathrm{FSym}_{nr}^0$$

with

$$\begin{array}{ll} al: & \text{ArrayList} \\ arg: & \texttt{int} \\ c: & \texttt{integer} \\ data: & \text{ArrayList} \to \text{Object}[\,] \\ i: & \texttt{int} \\ j: & \texttt{short} \\ length: & \text{ArrayList} \to \texttt{int} \\ para1: & \text{ArrayList} \\ sal: & \text{ArrayList} \\ v: & \texttt{int} \end{array}$$

and

$$\mathrm{PSym}_r = \mathrm{PSym}_r^0$$

Note 3.6. In the KeY system the user never has to explicitly define the whole type hierarchy and signature but the system automatically derives large parts of both from the JAVA CARD program under consideration. Only types and symbols that do not appear in the program must be declared manually. Note, however, that from a logical point of view the type hierarchy and the signature are fixed *a priori*, and formulae (and thus programs in modal operators being part of a formula) must only contain types and symbols declared in the type hierarchy and signature.

The syntactic categories of first-order logic are terms and formulae. Here, we need an additional category called *updates* [Beckert, 2001], which are used to (syntactically) represent state changes.

In contrast to first-order logic, the definition of terms and formulae (and also updates) in JAVA CARD DL cannot be done separately, since their definitions are mutually recursive. For example, a formula may contain terms which may contain updates. Updates in turn may contain formulae (see Example 3.9). Nevertheless, in order to improve readability we give separate definitions of updates, terms, and formulae in the following.

3.2.2 Syntax of Java Card DL Terms

Definition 3.7 (Terms of Java Card DL). *Given a Java Card DL signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*, the system* $\{Terms_A\}_{A\in\mathcal{T}}$ *of sets of* terms of static type A *is inductively defined as the least system of sets such that:*

- $x \in Terms_A$ *for all variables* $x{:}A \in \mathrm{VSym}$*;*
- $f(t_1, \ldots, t_n) \in Terms_A$ *for all function symbols* $f : A_1, \ldots, A_n \to A$ *in* $\mathrm{FSym}_r \cup \mathrm{FSym}_{nr}$ *and terms* $t_i \in Terms_{A_i'}$ *with* $A_i' \sqsubseteq A_i$ $(1 \le i \le n)$*;*
- (if ϕ then t_1 else t_2) $\in Terms_A$ *for all* $\phi \in Formulae$ *(⇒ Def. 3.14) and all terms* $t_1 \in Terms_{A_1}, t_2 \in Terms_{A_2}$ *with* $A = A_1 \sqcup A_2$*;*
- (ifExMin $x.\phi$ then t_1 else t_2) $\in Terms_A$ *for all variables* $x \in \mathrm{VSym}$*, all formulae* $\phi \in Formulae$ *(⇒ Def. 3.14), and all terms* $t_1 \in Terms_{A_1}$*,* $t_2 \in Terms_{A_2}$ *with* $A = A_1 \sqcup A_2$*;*
- $\{u\}\, t \in Terms_A$ *for all updates* $u \in Updates$ *(⇒ Def. 3.8) and all terms* $t \in Terms_A$*.*

In the style of Java Card syntax we often write $t.f$ *instead of* $f(t)$ *and* $a[i]$ *instead of* $[\,](a, i)$*.*[1]

Terms in Java Card DL play the same role as in first-order logic, i.e., they denote elements of the domain. The syntactical difference to first-order logic is the existence of terms of the form (if ϕ then t_1 else t_2) and (ifExMin $x.\phi$ then t_1 else t_2) (which could be defined for first-order logic as well). Informally, if ϕ holds, a conditional term (if ϕ then t_1 else t_2) denotes the domain element t_1 evaluates to. Otherwise, if ϕ does not hold, t_2 is evaluated. The meaning of a term (ifExMin $x.\phi$ then t_1 else t_2) is a bit more involved. If there is some d such that ϕ holds, then the whole term evaluates to the value denoted by t_1 under the variable assignment $\beta_x^{d'}$, where d' is the least element satisfying ϕ. Otherwise, if ϕ does not hold for any x, then t_2 is evaluated.

Terms can be prefixed by updates, which we define next.

3.2.3 Syntax of Java Card DL Updates

Definition 3.8 (Syntactic updates of Java Card DL). *Given a Java Card DL signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$*, the set Updates of* syntactic updates *is inductively defined as the least set such that:*

Function update: $(f(t_1, \ldots, t_n) := t) \in Updates$ *for all terms* $f(t_1, \ldots, t_n) \in Terms_A$ *(⇒ Def. 3.7) with* $f \in \mathrm{FSym}_{nr}$ *and* $t \in Terms_{A'}$ *s.t.* $A' \sqsubseteq A$*;*
Sequential update: $(u_1\,;u_2) \in Updates$ *for all* $u_1, u_2 \in Updates$*;*
Parallel update: $(u_1 \,||\, u_2) \in Updates$ *for all* $u_1, u_2 \in Updates$*;*

[1] Note, that $[\,]$ is a normal function symbol declared in the signature.

Quantified update: ($\texttt{for}\ x;\ \phi;\ u$) $\in$ *Updates for all* $u \in$ *Updates,* $x \in$ VSym*, and* $\phi \in$ *Formulae (⇒ Def. 3.14);*
Update application: ($\{u_1\}\ u_2$) $\in$ *Updates for all* $u_1, u_2 \in$ *Updates.*

Syntactic updates can be seen as a language for describing program transitions. Informally speaking, function updates correspond to assignments in an imperative programming language and sequential and parallel updates correspond to sequential and parallel composition, respectively. Quantified updates are a generalisation of parallel updates. A quantified update ($\texttt{for}\ x;\ \phi;\ u$) can be understood as (the possibly infinite) sequence of updates

$$\cdots \,||\, [x/t_n]u \,||\, \cdots \,||\, [x/t_0]u$$

put in parallel. The individual updates $[x/t_n]u, \ldots, [x/t_0]u$ are obtained by substituting the free variable x in the update u with all terms $t_n, \ldots, t_0$ such that $[x/t_i]\phi$ holds (it is assumed that all terms t_i evaluate to different domain elements). For parallel updates, the order matters. In case of a clash, i.e., if two updates put in parallel modify the same location, the latter one dominates the earlier one (if read from left to right). Coming back to our approximation of quantified updates by parallel updates, this means, that the order of the updates $[x/t_i]u$ put in parallel is crucial. As we see in Def. 3.27, the order depends on a total order $\preceq$, that is imposed on the domain, such that for all $[x/t_i]u$ the following holds: t_i evaluates to a domain element that is less than all the elements t_j ($j > i$) evaluate to (with respect to $\preceq$).

Updates vs. Other State Transition Languages

The idea of describing state changes by a (syntactically quite restrictive) language like JAVA CARD DL updates is not new and appears in slightly different ways in other approaches as well. For example, abstract state machines (ASMs) [Gurevich, 1995] are also based on updates which, however, have a different clash resolution strategy (clashing updates have no effect).

Another concept that is similar to updates are generalised substitutions in the B language [Abrial, 1996].

According to the semantics we define below, JAVA CARD DL terms are (like first-order terms) evaluated in a first-order interpretation that fixes the meaning of function and predicate symbols. However, in JAVA CARD DL models, we have many first-order interpretations (representing program states) rather than only one. Programs occurring in modal operators describe a state transition to the state in which the formula following the modal operator is evaluated. Updates serve basically the same purpose, but they are simpler in many respects.

A simple function update describes a transition from one state to exactly one successor state (i.e., the update process always "terminates normally" in our terminology). Exactly one "memory location" is changed during this transition. None of the above holds in general for a JAVA assignment. Furthermore,

the syntax of updates generated by the calculus that we define (⇒ Sect. 3.6) is restricted even further, making analysis and simplification of state change effects easier and efficient. Updates (together with case distinctions) can be seen as a normal form for programs and, indeed, the idea of our calculus is to stepwise transform a program to be verified into a sequence of updates, which are then simplified and applied to first-order formulae.

Example 3.9. Given the type hierarchy and the signature from Examples 3.3 and 3.5, respectively, the following are JAVA CARD DL terms:

n	a variable
c	a non-rigid 0-ary function (constant)
$\{c := 0\}(c)$	a term with a function update
$\{c := 0 \,\|\|\, c := 1\}(c)$	a term with a parallel update
$\{\texttt{for}\ x;\ x \doteq 0 \mid x \doteq 1;\ c := x\}(c)$	a term with a quantified update
$\{\texttt{for}\ a;\ a \sqsubseteq \text{ArrayList};\ \mathit{length}(x) := 0\}(\mathit{length}(al))$	a term with a quantified update

In contrast, the following are *not* terms:

f	wrong number of arguments
$\{n := 0\}(c)$	update tries to change the value of a variable
$\{g := 0\}(c)$	update tries to change the value of a rigid function symbol
$\{\texttt{for}\ i;\ i \doteq 0 \mid i \doteq 1;\ c := i\}(c)$	an update quantifying over a term instead of a variable (i was declared to be a function symbol)

Updates vs. Substitutions

In classical dynamic logic [Harel et al., 2000] and Hoare logic [Hoare, 1969] there are no updates. Modifications of states are expressed using equations and syntactic substitutions. Consider, for example, the following instance of the assignment rule for classical dynamic logic

$$\frac{\Gamma,\ x' \doteq x + 1 \Longrightarrow [x/x']\phi,\ \Delta}{\Gamma \Longrightarrow \langle x = x + 1\rangle\phi,\ \Delta}$$

where the fresh variable x' denotes the new value of x. The formula ϕ must be evaluated with the new value x' and therefore x is substituted with x'. The equation $x' \doteq x + 1$ establishes the relation between the old and the new value of x.

In principle, updates are not more expressive than substitutions. However, for reasoning about programs in an object-oriented programming language like JAVA CARD updates have some advantages.

The main advantage of updates is that they are part of the syntax of the (object-level) logic, while substitutions are only used on the meta-level

to describe and manipulate formulae. Thus, updates can be collected and need not be applied until the whole program has been symbolically executed (and, thus, has disappeared). The collected updates can be simplified before they are actually applied, which often helps to avoid case distinctions in proofs. Substitutions in contrast are applied immediately and thus there is no chance of simplification (for more details see Sect. 3.6.1).

Another point in favour for updates is that substitutions—as usually defined for first-order logic—replace variables with terms. This, however, does not help to handle JAVA CARD assignments since they modify non-rigid functions rather than variables. Thus, in order to handle assignments with updates, a more general notion of substitution would become necessary.

3.2.4 Syntax of JAVA CARD DL Formulae

Before we define the syntax of JAVA CARD DL formulae, we first define *normalised* JAVA CARD programs, which are allowed to appear in formulae (within modalities). The normal form can be established automatically by a simple program transformation and/or extension of the type hierarchy and the signature and does not constitute a real restriction.

Normalised JAVA CARD Programs

The definition of normalised JAVA CARD programs is necessary for two reasons. First, we do not want to handle certain features of JAVA CARD (like, e.g., inner classes) in the calculus (⇒ Sect. 3.4.6), because including them would require many rules to be added to our calculus. The approach we pursue is to remove such features by a simple program transformation. The second reason is that a JAVA CARD program must only contain types and symbols declared in the type hierarchy and signature. Note, that this does not restrict the set of possible JAVA CARD programs. It is always possible to adjust the type hierarchy and signature to a given program.

Since JAVA CARD code can appear in formulae, we actually have to give a formal definition of the syntax of JAVA CARD. This however goes beyond the scope of this book and we refer the reader to the JAVA CARD language specification [Chen, 2000, Sun, 2003d,c].

Definition 3.10 (Normalised JAVA CARD programs). *Given a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *for a signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$*, a* normalised JAVA CARD program P *is a set of (abstract) class and interface definitions satisfying the following constraints:*

1. *P is compile-correct and compile-time constants of an integer type do not cause overflow.*[2]

[2] The second condition can be checked statically. For example, a compile time constant like `final byte b=(byte)500;` is not allowed since casting the literal `500` of type `int` to type `byte` causes overflow.

2. *P does not contain inner classes.*
3. *Identifiers in declarations of local variables, attributes, and parameters of methods (and constructors) are unique.*
4. *$A \in \mathcal{T}_a$ for all interface and abstract class types A declared in or imported into P.*
5. *$A \in \mathcal{T}_d$ for all non-abstract class types A declared in or imported into P.*
6. *$C \sqsubseteq D$ iff C is implicitly or explicitly declared as a subtype of D (using the keywords* `extends` *or* `implements`*), for all (abstract) class or interface types C, D declared in or imported into P.*
7. *For all array types $A\underbrace{[]\cdots[]}_{n \text{ times}}$ (or $A[]^n$ for short where $A[]^0 = A$) occurring in P and $1 \leq i \leq n$:*
   - *$A \in \mathcal{T}$,*
   - *$B[]^m \sqsubseteq A[]^n$ iff $B[]^{m-1} \sqsubseteq A[]^{n-1}$ for all $B[]^m \in \mathcal{T}$ ($m \geq 1$),*
   - *$A[]^i \in \mathcal{T}_d$,*
   - *$A[]^i \sqsubseteq$ Object,*
   - *$A[]^i \sqsubseteq$ Serializable,*
   - *$A[]^i \sqsubseteq$ Cloneable, and*
   - *$B \not\sqsubseteq A[]^i \in \mathcal{T}_d$ for all non-array types $B \in \mathcal{T} \setminus \{\bot, \text{Null}\}$,*
   - *$\text{Null} \sqsubseteq A[]^i$.*
8. *For all local variables and static field declarations "A id;" in P:*
   a) If A is not an array type, then id:$A \in \text{FSym}_{nr}$.
   b) If $A = A'[]^n$ is an array type, then id:$(A'[]^n) \in \text{FSym}_{nr}$.
9. *For all non-static field declarations "A id;" in a class C in P:*
   a) If A is not an array type, then $id : (C \to A) \in \text{FSym}_{nr}$.
   b) If $A = A'[]^n$ is an array type, then $id : (C \to A'[]^n) \in \text{FSym}_{nr}$.

Let Π denote the set of all normalised Java Card *programs.*

Not surprisingly, we require that the programs P we consider are compile-correct (Constraint (1)). Constraint (2) requires that P does not contain inner classes like, e.g., anonymous classes. In principle, this restriction could be dropped. This however would result in a bunch of extra rules for the calculus to be defined in Sect. 3.4.6.

In contrast to the programming language Java Card, in Java Card DL we do not have overloading of function or predicate symbols. Therefore we require that the identifiers used in declarations in P are unique (Constraint (3)). For instance, it is not allowed to have two local variables with the same name occurring in P. Field hiding is disallowed by the same token.

Note, that the Constraints (2) and (3) are harmless restrictions in the sense that any Java Card program can easily be transformed into an equivalent program satisfying the constraints.

Constraints (4) and (5) make sure that all non-array reference types declared in and imported into P are contained in the type hierarchy. This in particular applies to all classes that are automatically imported into any program like the classes in package `java.lang` (in particular `Object`).

Constraint (6) guarantees that the inheritance hierarchy of the JAVA CARD program P is correctly reflected by the subtype relation in the type hierarchy.

Array reference types are addressed in Constraint (7). Array types are not declared explicitly in JAVA CARD like class or interface types but nevertheless they still must be part of the type hierarchy and the subtype relation must match the inheritance hierarchy in JAVA CARD, i.e., array types are subtypes of Serializable, Cloneable, and Object.

The first condition requires the element type A of an array type $A[]^n$ to be part of the type hierarchy ($A \in \mathcal{T}$). The subtype relation between two array types $B[]^m$ and $A[]^n$ is recursively defined on the component types $B[]^{m-1}$ and $A[]^{n-1}$. Therefore, we additionally postulate that all array types up to dimension n with element type A are contained in the set of dynamic types ($A[]^i \in \mathcal{T}_d$) as well. Then the recursive definition is well-founded since eventually we arrive at non-array types and we require that $A[]^i \sqsubseteq \text{Object}$. Finally, we stipulate that non-array types $B \in \mathcal{T} \setminus \{\bot, \text{Null}\}$ must not be a subtype of any array type $A[]^i$.

Local variables and static fields in JAVA CARD occur as non-rigid 0-ary functions in the logic (i.e., as constants). Therefore, we require for any such element a corresponding function to be present in the signature (Constraint (8)).

Finally, in Constraint (9) we consider non-static fields which are represented by non-rigid unary functions that map instances of the class declaring the field to elements of the field type.

In order to normalise a JAVA CARD program, Constraints (2) and (3) can always be satisfied by performing a program transformation. For example, inner classes can be transformed into top-level classes; identifiers (e.g., attributes or local variables) can be renamed. On the other hand, meeting the Constraints (4)–(9) may require an extension of the underlying type hierarchy and signature, since only declared types and symbols may be used in a normalised JAVA CARD program. However, such an extension is harmless and is done automatically by the KeY system, i.e., the user does not have to explicitly declare all the types and symbols occurring in the JAVA CARD program to be considered.

Example 3.11. Given the type hierarchy and signature from Examples 3.3 and 3.5, respectively, the following set of classes and interfaces constitute a normalised JAVA CARD program.

JAVA

```
abstract class AbstractCollection {
}

interface List {
}

abstract class AbstractList
  extends AbstractCollection implements List {
```

```
}

class ArrayList extends AbstractList {

  static ArrayList sal;
  static int v;

  Object[] data;
  int length;

  public static void demo1() {
    int i=0;
  }

  public static void demo2(ArrayList para1) {
    // int i=1; violates Constraint (3)
    short j;
    if (para1==null)
      j=0;
    else
      j=1;
    para1.demo3();
  }

  void demo3() {
    this.length=this.length+1;
  }

  int inc(int arg) {
    return arg+1;
  }

}

// class Violate { violates Constraint (5)
//   int k; violates Constraint (8)
// }
```

JAVA

The above program satisfies all the constraints from Def. 3.10. All interface types, (abstract) class types, and array types are contained in the corresponding JAVA CARD DL type hierarchy, and for all identifiers in the program there is a type correct function in the signature.

The statement in line 24 (commented out) would violate Constraint (3) since method `demo1` already declares a local variable with identifier `i`.

Similarly, declaring a class type `Violate` that is not contained in the type hierarchy (as in line 43) violates Constraint (5). Also not allowed is declaring a local variable if the signature does not contain the corresponding function symbol (line 44).

Within modal operators we exclusively allow for sequences of statements. A so-called *program statement* is either a normal JAVA statement, a *method-body statement*, or a *method-frame statement*. Note that logical variables, in contrast to non-rigid function symbols reflecting local program variables, attributes, and arrays, must not occur in programs.

Intuitively, a method-body statement is a shorthand notation for the precisely identified implementation of method $m(\ldots)$ in class T. That is, in contrast to a normal method call in JAVA CARD 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.

A method-frame statement is required when handling a method call by syntactically replacing it with the method's implementation (⇒ Sect. 3.6.5). To handle the return statement in the right way, it is necessary

1. to record the object field or variable x that the result is to be assigned to, and
2. to mark the boundaries of the implementation body when it is substituted for the method call.

For that purpose, we allow a method-frame statement to occur as a JAVA CARD DL program statement.

Definition 3.12 (JAVA CARD DL program statement). *Let $P \in \Pi$ be a normalised JAVA CARD program. Then a* JAVA CARD DL program statement *is*

- *a JAVA statement as defined in the JAVA language specification [Gosling et al., 2000, § 14.5] (except* `synchronized`*),*
- *a* method-body statement

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

  where
  - *$\mathit{target}\texttt{.}m\texttt{(}t_1\texttt{,}\ldots\texttt{,}t_n\texttt{)}$ is a method invocation expression,*
  - *the type T points to a class declared in P (from which the implementation is taken),*
  - *the result of the method is assigned to retvar after return (if the method is not void), or*
- *a* method-frame statement

  `method-frame(result->`*retvar*`, source=`T`, this=`*target*`) : { `*body*` }`

  where

- *the return value of the method is assigned to retvar when body has been executed (if the method is not void),*
- *the type T points to the class in P providing the particular method implementation,*
- *target is the object the method was invoked on,*
- *body is the body of the invoked method.*

Thus, all JAVA statements that are defined in the official language specification can be used (except for `synchronized` blocks), and there are two additional ones: a `method-body` statement and a `method-frame` statement.

Another extension is that we do not require *definite assignment*. In JAVA, the value of a local variable or `final` field must have a definitely assigned value when any access of its value occurs [Gosling et al., 2000, § 16]. In JAVA CARD DL we allow sequences of statements that violate this condition (the variable then has a well-defined but unknown value).

Note, that the additional constructs and extensions are a "harmless" extension as they are only used for proof purposes and never occur in the verified JAVA CARD programs.

Definition 3.13 (Legal sequence of JAVA CARD DL program statements). *Let $P \in \Pi$ be normalised JAVA CARD program.*

A sequence $st_1 \cdots st_n$ ($n \geq 0$) of JAVA CARD DL program statements is legal *w.r.t. to P if P enriched with the class declaration*

```
public class DefaultClass {
  public static void defaultMethod() {
    st_1
    ⋮
    st_n
  }
}
```

where DefaultClass and defaultMethod are fresh identifiers—is a normalised JAVA CARD program, except that $st_1 \cdots st_n$ do not have to satisfy the definite assignment *condition [Gosling et al., 2000, § 16].*

Now we can define the set of JAVA CARD DL formulae:

Definition 3.14 (Formulae of JAVA CARD DL). *Let a signature* (VSym, $\mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha$) *for a type hierarchy T a normalised JAVA CARD program $P \in \Pi$ be given.*

Then, the set Formulae of JAVA CARD DL formulae *is inductively defined as the least set such that:*

- $r(t_1, \ldots, t_n) \in$ *Formulae for all predicate symbols* $r : A_1, \ldots, A_n \in \mathrm{PSym}_r \cup \mathrm{PSym}_{nr}$ *and terms* $t_i \in Terms_{A_i'}$ *(⇒ Def. 3.7) with* $A_i' \sqsubseteq A_i$ *(*$1 \leq i \leq n$*),*

- true, false ∈ *Formulae,*
- $!\,\phi,\ (\phi \mid \psi),\ (\phi\ \&\ \psi), (\phi \texttt{ -> } \psi),\ (\phi \texttt{ <-> } \psi) \in$ *Formulae for all* $\phi, \psi \in$ *Formulae,*
- $\forall x.\phi,\ \exists x.\phi \in$ *Formulae for all* $\phi \in$ *Formulae and all variables* $x \in$ VSym*,*
- $\{u\}\ \phi \in$ *Formulae for all* $\phi \in$ *Formulae and* $u \in$ *Updates (⇒ Def. 3.8),*
- $\langle p\rangle\phi,\ [p]\phi \in$ *Formulae for all* $\phi \in$ *Formulae and any legal sequence* p *of* Java Card *DL program statements.*

In the following we often abbreviate formulae of the form $(\phi \texttt{ -> } \psi)$ & $(!\,\phi \texttt{ -> } \xi)$ *by* if ϕ then ψ else ξ.

Example 3.15. Given the type hierarchy and the signature from Examples 3.3 and 3.5, respectively, and the normalised Java Card DL program from Example 3.11, the following are Java Card DL formulae:

$\{c := 0\}(c \doteq 0)$	a formula with an update
$(\{c := 0\}c) \doteq c$	a formula containing a term with an update
`sal !=` $\doteq$ `null -> ⟨ArrayList.demo2(sal);⟩j` $\doteq 1$	a formula with a modal operator
$\{$`sal` $:=$ `null`$\}\langle$`ArrayList.demo2(sal);`$\rangle$`j` $\doteq 0$	a formula with a modal operator and an update
$\{$`v` $:= g\}\langle$`ArrayList al=new ArrayList(); v=al.inc(v);`$\rangle$`(v` $\doteq g+1)$	a formula with a modal operator and an update
$\{$`v` $:= g\}\langle$ `ArrayList al=new ArrayList(); v=al.inc(v)@ArrayList`$\rangle$`(v` $\doteq g+1)$	a method-body statement within a modal operator
$\langle$`int i=0; v=i;`$\rangle$`(v` $\doteq 0)$	local variable declaration and assignment within a modal operator

Note 3.16. In program verification, one is usually interested in proving that the program under consideration satisfies some property for all possible input values. Since, by definition, terms (except those declared as static fields) and in particular logical variables, i.e., variables from the set VSym, may not occur within modal operators, it can be a bit tricky to express such properties. For example, the following is not a syntactically correct Java Card DL formula:

$$\forall n.(\langle\ \texttt{ArrayList al=new ArrayList(); v=al.inc(}n\texttt{)}\rangle(\texttt{v} \doteq n+1))$$

To express the desired property, there are two possibilities. The first one is using an update to bind the program variable to the quantified logical variable:

$$\forall n.\{\texttt{v} := n\}(\langle\ \texttt{ArrayList al=new ArrayList(); v=al.inc(v);}\rangle(\texttt{v} \doteq n+1))$$

The second possibility is to use an equation:

$$\forall n.(n \doteq \texttt{v} \texttt{ -> } \langle \texttt{ ArrayList al=new ArrayList();}\\ \texttt{v=al.inc(v);}\rangle(\texttt{v} \doteq n+1))$$

Both possibilities are equivalent with respect to validity: the first one is valid iff the second one is valid.[3]

Before we define the semantics of JAVA CARD DL in the next section, we extend the definition of free variables from Chap. 2 to the additional syntactical constructs of JAVA CARD DL.

Definition 3.17. *We define the set $fv(u)$ of free variables of an update u by:*

- $fv(f(t_1,\ldots,t_n) := t) = fv(t) \cup \bigcup_{i=1}^{n} fv(t_i)$,
- $fv(u_1\,;u_2) = fv(u_1) \cup fv(u_2)$,
- $fv(u_1 \,||\, u_2) = fv(u_1) \cup fv(u_2)$,
- $fv(\texttt{for}\ x;\ \phi;\ u) = (fv(\phi) \cup fv(u)) \setminus \{x\}$.

For terms and formulae we extend Def. 2.18 as follows:

- $fv(\text{if}\ \ \phi\ \ \text{then}\ \ t_1\ \ \text{else}\ \ t_2) = fv(\phi) \cup fv(t_1) \cup fv(f_2)$
- $fv(\text{ifExMin}\, x.\phi\ \ \text{then}\ \ t_1\ \ \text{else}\ \ t_2) = ((fv(\phi) \cup fv(t_1)) \setminus \{x\}) \cup fv(f_2)$
- $fv(\{u\}t) = fv(u) \cup fv(t)$ *for a term* t,
- $fv(\{u\}\phi) = fv(u) \cup fv(\phi)$ *for a formula* ϕ,
- $fv(\langle p\rangle\phi) = fv(\phi)$ *for a formula* ϕ,
- $fv([p]\phi) = fv(\phi)$ *for a formula* ϕ.

3.3 Semantics

We have seen that the syntax of JAVA CARD DL extends the syntax of first-order logic with updates and modalities. On the semantic level this is reflected by the fact that, instead of one first-order model, we now have an (infinite) set of such models representing the different program states. Traditionally, in modal logics the different models are called *worlds*. But here we call them *states*, which better fits the intuition.

Our semantics of JAVA CARD DL is based on so-called *Kripke structures*, which are commonly used to define the semantics of modal logics. In our case a Kripke structure consists of

- a partial first-order model $\mathcal{M}$ fixing the meaning of *rigid* function and predicate symbols,
- an (infinite) set $\mathcal{S}$ of states where a state is any first-order model refining $\mathcal{M}$, thus assigning meaning to the non-rigid function and predicate symbols (which are not interpreted by $\mathcal{M}$), and

[3] Please note that both formulae ϕ_1, ϕ_2 are not logically equivalent in the sense that $\phi_1 \leftrightarrow \phi_2$ is logically valid.

- a program relation ρ fixing the meaning of programs occurring in modalities: $(S_1, p, S_2) \in \rho$ iff the sequence p of statements when started in state S_1 terminates in state S_2, assuming it is executed in some static context, i.e., in some static method declared in some public class.

3.3.1 Kripke Structures

Definition 3.18 (Java Card DL Kripke structure). *Let a signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$ *be given and let* $P \in \Pi$ *be a normalised Java Card program.*

A Java Card DL Kripke structure $\mathcal{K}$ *for that signature, type hierarchy, and program is a tuple* $(\mathcal{M}, \mathcal{S}, \rho)$ *consisting of a partial first-order model* $\mathcal{M} = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$*, a set* $\mathcal{S}$ *of states, and a program relation* ρ *such that:*

- $\mathcal{T}_0 = \mathcal{T}$*;*
- *the partial domain* $\mathcal{D}_0$ *is a set satisfying*
  - $\mathbb{Z} = \mathcal{D}_0^{\texttt{integerDomain}}$,
  - $\{tt, ff\} = \mathcal{D}_0^{\mathrm{boolean}}$,
  - $\{null\} = \mathcal{D}_0^{\mathrm{Null}}$,
  - *for all dynamic types* $A \in \mathcal{T}_d \setminus \{\mathrm{Null}\}$ *with* $A \sqsubseteq \mathrm{Object}$ *there is a countably infinite set* $\mathcal{D}_0' \subseteq \mathcal{D}_0$ *such that* $\delta_0(d) = A$ *for all* $d \in \mathcal{D}_0'$,
  - *for all* $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}_r \cup \mathrm{FSym}_{nr}$

$$D_0(f) = \begin{cases} \emptyset & \text{if } f \in \mathrm{FSym}_{nr} \\ \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n} & \text{if } f \in \mathrm{FSym}_r \end{cases}$$

  - *for all* $p : A_1, \ldots, A_n \in \mathrm{PSym}_r \cup \mathrm{PSym}_{nr}$

$$D_0(p) = \begin{cases} \emptyset & \text{if } p \in \mathrm{PSym}_{nr} \\ \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n} & \text{if } p \in \mathrm{FSym}_r \end{cases}$$

  - $\mathcal{I}_0(f)$ *for* $f \in \mathrm{FSym}_r^0$ *(see App. A.2.1),*
  - $\mathcal{I}_0(p)$ *for* $p \in \mathrm{PSym}_r^0$ *(see App. A.2.2);*
- *the set* $\mathcal{S}$ *of Java Card DL states consists of all first-order models* $(\mathcal{D}, \delta, \mathcal{I})$ *refining* $\mathcal{M}$ *with*
  - $\mathcal{D} = \mathcal{D}_0$,
  - $\delta = \delta_0$*;*
- *the program relation* ρ *is, for all states* $S_1, S_2 \in \mathcal{S}$ *and any legal sequence* p *of Java Card DL program statements, defined by:*

$$\rho(S_1, p, S_2)$$
iff
p started in S_1 *in a static context terminates normally in* S_2
according to the Java language specification [Gosling et al., 2000].

The partial model $\mathcal{M}$ is called Kripke seed *since it determines the set of states of a* JAVA CARD *DL Kripke structure.*

Note 3.19. In the above definition we require that the partial domain $\mathcal{D}_0$ (which is equal to the domain $\mathcal{D}$) is a set satisfying the mentioned properties. This guarantees in particular that the domain contains exactly the two elements *tt* and *ff* with dynamic type boolean and that *null* is the only element with dynamic type Null.

Moreover, we require that for each dynamic subtype A of type Object (except type Null) there is a countably infinite subset $\mathcal{D}_0' \subseteq \mathcal{D}_0$ with $\delta_0(d) = A$. These domain elements represent the JAVA CARD objects of dynamic type A. Objects can be created dynamically during the execution of a JAVA CARD program and therefore we do not know the exact number of objects in advance. Since for a smooth handling of quantifiers in a calculus it is advantageous to have a constant domain for all JAVA CARD DL states (see below), we cannot extend the domain on demand if a new object is created. Therefore, we simply require an infinite number of domain elements with an appropriate dynamic type making sure that there is always an unused domain element available to represent a newly created object.

Constant-domain Assumption

Def. 3.18 requires that both the domain and the dynamic type function are the same for all states in a Kripke structure. This so-called *constant-domain assumption* is a harmless but reasonable and useful restriction as the following example shows.

If we assume a constant domain, then the formula

$$\forall x.p(x) \mathrel{-\!\!>} \langle\pi\rangle\forall x.p(x) \ ,$$

where p is a rigid predicate symbol, is valid in all Kripke structures because p cannot be affected by the program π. Without the constant domain assumption there could be states where this formula does not hold since the domain of the state reached by π may have more elements than the state where $\forall x.p(x)$ holds and in particular might include elements for which p does not hold.

A problem similar to the one above already appears in classical modal logics. In this setting constant-domain Kripke structures are characterised by the so-called *Barcan formula* $\forall x.\Box p(x) \rightarrow \Box\forall x.p(x)$ (see, e.g., the book by Fitting and Mendelsohn [1999]).

The Kripke seed $\mathcal{M}$ of a Kripke structure (Def. 3.18) fixes the interpretation of the rigid function and predicate symbols, i.e., of those symbols that have the same meaning in all states. Moreover, it is "total" for these symbols in the sense that it assigns meaning to rigid symbols for all argument tuples, i.e., $D_0(s) = \mathcal{D}_0^{A_1} \times \cdots \times \mathcal{D}_0^{A_n}$ for any rigid function or predicate symbol s.

That means, for example, that division by zero is defined, i.e., $\mathcal{I}_0(x/y)$ (we use infix notation for better readability) yields some (fixed but unknown) element $d \in \mathbb{Z} \subseteq \mathcal{D}_0$ for $y = 0$.

Handling Undefinedness

The way we deal with undefinedness is based on *underspecification* as proposed by Gries and Schneider [1995], Constable and O'Donnell [1978]. Hähnle [2005] argues that this approach is superior to other approaches (at least in the context of specification and verification of programs).

The basic idea is that any function f that is undefined for certain argument tuples (like, e.g., / which is undefined for $\{(x, 0) \mid x \in \mathbb{Z}\}$) is made total by assigning a fixed but unknown result value for those arguments where it is undefined. This is achieved using a dedicated (semantic) choice function $choice_f$ which has the same arity as f. For example, for / the choice function $choice_/$ could be defined as $choice_/(x, 0) = x$.

In the presence of choice functions, the definition of validity needs to be revised such that a formula ϕ is said to be valid in a model $\mathcal{M}$ iff it is valid in $\mathcal{M}$ for all possible definitions of the choice functions. That is, it is crucial that all possibilities for a choice function are considered rather than relying on just one particular possibility.

In Example 2.41 we explained how this can be achieved in classical first-order logic making use of partial models, leaving open the interpretation of functions for critical argument tuples. A formula is then valid in a (partial) model $\mathcal{M}$ iff it is valid in all (total) models refining $\mathcal{M}$—making sure that all possibilities for choice functions are considered.

In order to carry over this approach to the Kripke semantics of JAVA CARD DL there are two options:

1. Leaving open the interpretation of functions for critical argument tuples in the Kripke seed. Then all possibilities for the choice functions are considered in the states of the Kripke structure, which are defined as the set of all models refining the Kripke seed.
2. Fixing a particular choice function in the Kripke seed. Then in order to consider all choice functions all possible Kripke seeds need be to taken into account.

In Def. 3.18 we chose the second of these two options, and there are good reasons for this decision. Since a formula is defined to be valid iff it is valid in all Kripke structures ($\Rightarrow$ Def. 3.38), we need to consider all Kripke structures (and thus all Kripke seeds) anyway. The second and more important argument is that, if we chose the first option, each single Kripke structure contains states in which the same "undefined" term would evaluate to different values. Such a term would then not be rigid anymore (Lemma. 3.33)—even if the function symbol is declared to be rigid and the arguments are rigid. That would heavily complicate the definitions of the semantics of JAVA CARD DL formulae containing modal operators and of update simplification in Sect. 3.9. For example, without modifying the

semantics of JAVA CARD DL formulae, the formula

$$g \doteq 5/0 \mathrel{-\!\!>} \langle \texttt{int i=0;} \rangle g \doteq 5/0 \ ,$$

where g is a rigid constant, would no longer be valid since the program might terminate in a state where $5/0$ has a meaning different from that in the initial state (whereas g has the same meaning in all states since it is rigid). Hence it is beneficial to fix the semantics of all rigid functions for all argument tuples already in the Kripke seed.

Please note, that—besides underspecification—there are several other ways to deal with undefinedness in formal languages. One possibility is to introduce an explicit value *undefined*. That approach is pursued, e.g., in OCL [OCL 2.0]. It has the disadvantage that the user needs to know non-standard semantics in order to evaluate expressions. Further approaches, such as allowing for partially defined functions, are discussed in the article by Hähnle [2005].

The Kripke seed does not provide an interpretation of the non-rigid symbols, which is done by the models refining the seed, i.e., the states of the Kripke structure.

The semantics of normalised JAVA CARD programs is given by the relation ρ, where ρ holds for (S_1, p, S_2) iff the sequence p of statements, when started in S_1, terminates *normally* in S_2. *Normal* termination means that the program does not terminate abruptly (e.g., because of an uncaught exception). Otherwise ρ does not hold, i.e., if the program terminates *abruptly* or does not terminate at all.

Non-reachable States

According to Def. 3.18, the set S of JAVA CARD DL states contains *all* possible states (i.e., all structures refining the Kripke seed). That implies that a JAVA CARD DL Kripke structure also contains states that are not reachable by any JAVA CARD program (e.g., states in which a class is both marked as initialised and erroneous). The main reason for not excluding such states is that even if they cannot be reached by any JAVA CARD program, they can still be reached (or better: described) by updates. For example, the update

$$T\texttt{.<classInitialised>} := \text{TRUE} \parallel T\texttt{.<erroneous>} := \text{TRUE}$$

describes a state in which a class $T \sqsubseteq$ Object is both initialised and erroneous. Such states do not exist in JAVA CARD.

There is no possibility to syntactically restrict the set of updates such that only states reachable by JAVA CARD programs can be described. Of course, one could modify the semantics of updates such that updates leading to non-reachable states do not terminate or yield an unspecified state. That however would make the semantics and simplification of updates much more complicated.

Note 3.20. The definition that abrupt termination and non-termination are treated the same is not a necessity but a result of the answer to the question of when we consider a program to be correct. On the level of Java Card DL we say that a program is totally correct if it terminates normally and if it satisfies the postcondition (assuming it satisfies the precondition). Thus, if something unexpected happens and the program terminates abruptly then it is not considered to be totally correct—even if the postcondition holds in the state in which the execution of the program abruptly stops.

Other languages like, e.g., the Java Modeling Language (JML) have a more fine-grained interpretation of correctness with respect to (abrupt) termination. JML distinguishes between normal termination and abrupt termination by an uncaught exception, and it allows to specify different postcondition for each of the two cases. Since in the KeY tool we translate JML expressions into Java Card DL formulae we somehow have to mimic the distinction between non-termination and abrupt termination in Java Card DL.

This is done by performing a program transformation such that the resulting program catches all exceptions at top-level and thus always terminates normally. The fact, that the original program would have terminated abruptly is indicated by the value of a new Boolean variable. For example, in the following formula, the program within the modal operator terminates normally, independently of the value of `j`.

KeY

```
\<{
Throwable thrown = null;
  try {
    i = i / j;
 } catch (Exception e) {
   thrown = e;
 }
}\> (thrown != null)
```

KeY

In the postcondition the formula `thrown` $\doteq$ `null` holds if and only if the original program (without the `try-catch` block) terminates abruptly.

For a more detailed account of this issue the reader is referred to Sects. 3.7.1 and 8.2.3.

Analogously to the syntax definition, the semantics of Java Card DL updates, terms, and formulae is defined mutually recursive. For better readability we ignore this fact and give separate definitions for the semantics of update, terms, and formulae, respectively.

3.3.2 Semantics of Java Card DL Updates

Similar to the first-order case we inductively define a valuation function $\mathrm{val}_{\mathcal{M}}$ assigning meaning to updates, terms, and formulae. Since non-rigid function

and predicate symbols can have different meanings in different states, the valuation function is parameterised with a JAVA CARD DL state, i.e., for each state $\mathcal{S}$, there is a separate valuation function.

The intuitive meaning of updates is that the term or formula following the update is to be evaluated not in the current state but in the state described by the update. To be more precise, updates do not describe a state completely, but merely the difference between the current state and the target state. As we see later this is similar to the semantics of programs contained in modal operators and indeed updates are used to describe the effect of programs.

In parallel updates $u_1 \parallel u_2$ (as well as in quantified updates) clashes can occur, where u_1 and u_2 simultaneously modify a non-rigid function f for the same arguments in an inconsistent way, i.e., by assigning different values. To handle this problem, we use a *last-win* semantics, i.e., the update that syntactically occurs last dominates earlier ones. In the more general situation of quantified (unbounded parallel) updates `for` x; ϕ; u, we assume that a fixed well-ordering $\preceq$ on the universe $\mathcal{D}$ exists (i.e., a total ordering such that every non-empty subset $\mathcal{D}_{sub} \subseteq \mathcal{D}$ has a least element $min_{\preceq}(\mathcal{D}_{sub})$). The parallel application of unbounded sets of updates can then be well-ordered as well, and clashes can be resolved by giving precedence to the update assigning the smallest value. For this reasons, we first equip JAVA CARD DL Kripke structures with a well-ordering on the domain.

Definition 3.21 (JAVA CARD DL Kripke structure with ordered domain). *A* JAVA CARD DL Kripke structure with ordered domain $\mathcal{K}_{\preceq}$ *is a JAVA CARD DL Kripke structure* $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ *with a well-ordering on* $\mathcal{D}$*, i.e., a binary relation* $\preceq$ *with the following properties:*

- $x \preceq x$ *for all* $x \in \mathcal{M}$ *(reflexivity),*
- $x \preceq y$ *and* $y \preceq x$ *implies* $x = y$ *(antisymmetry),*
- $x \preceq y$ *and* $y \preceq z$ *implies* $x \preceq z$ *(transitivity), and*
- *any non-empty subset* $\mathcal{D}_{sub} \subseteq \mathcal{D}$ *has a least element* $min_{\preceq}(\mathcal{D}_{sub})$*, i.e.,* $min_{\preceq}(\mathcal{D}_{sub}) \preceq y$ *for all* $y \in \mathcal{D}_{sub}$ *(well-orderedness).*

As every set can be well-ordered (based on Zermelo-Fraenkel set theory [Zermelo, 1904]), this does not restrict the range of possible domains.

The particular order imposed on the domain of a Kripke structure is a parameter that can be chosen depending on the problem. In the implementation of the KeY system, we have chosen the following order as it allows to capture the effects of a particular class of loops in quantified updates in a rather nice way Gedell and Hähnle [2006]. Note however, that the order can be modified without having to adapt other definitions of the logic except for the predicate *quanUpdateLeq* that allows to access the order on the object level (it is required for update simplification (⇒ Sect. 3.9)).

Definition 3.22 (KeY JAVA CARD DL Kripke structure). *A* KeY JAVA CARD DL Kripke structure *is a JAVA CARD DL Kripke structure with ordered domain, where the order* $\preceq$ *is defined for any* $x, y \in \mathcal{D}^{\top}$ *as follows:*

- *If* $\delta_0(x) \neq \delta_0(y)$ *then* $\begin{cases} x \preceq y & \text{if } \delta_0(x) \sqsubseteq \delta_0(y) \\ y \preceq x & \text{if } \delta_0(y) \sqsubseteq \delta_0(x) \\ x \preceq y & \text{if } \delta_0(x) \leq_{lex} \delta_0(y) \text{ and neither} \\ & \delta_0(x) \sqsubseteq \delta_0(y) \text{ nor } \delta_0(y) \sqsubseteq \delta_0(x) \end{cases}$

  where $\leq_{lex}$ *is the usual lexicographic order on the names of types.*
- *If* $\delta_0(x) = \delta_0(y)$ *then*
  - *if* $\delta_0(x) =$ boolean *then* $x \preceq y$ *iff* $x =$ *ff*
  - *if* $\delta_0(x) =$ `integerDomain` *then* $x \preceq y$ *iff*
    $x \geq 0$ *and* $y < 0$ *or*
    $x \geq 0$ *and* $y \geq 0$ *and* $x \leq y$*, or*
    $x < 0$ *and* $y < 0$ *and* $y \leq x$
  - *if* Null $\neq A = \delta_0(x) \sqsubseteq$ Object *then* $x \preceq y$ *iff* $index_A(x) \preceq index_A(y)$ *where* $index_T : \mathcal{D}^T \to$ `integer` *is some arbitrary but fixed bijective mapping for all dynamic types* $T \in \mathcal{T}_d \setminus \{\text{Null}\}$
  - *if* $\delta_0(x) =$ Null *then* $x = y$.

The semantics of an update is defined—relative to a given JAVA CARD DL state—as a partial first-order model $(\mathcal{T}_0, \mathcal{D}_0, \delta, D, \mathcal{I}_0)$ that is defined exactly on those tuples of domain elements that are affected by the update, i.e., the partial model describes only the modifications induced by the update. Thus, the semantics of updates is given by a special class of partial models that differ only in D and $\mathcal{I}_0$ from a JAVA CARD DL state $(\mathcal{T}_0, \mathcal{D}_0, \delta, D', \mathcal{I}_0')$ (here seen as a partial model), i.e., an update neither modifies the set $\mathcal{T}_0$ of fixed types, nor the partial domain $\mathcal{D}_0$, nor the dynamic type function δ. In order to improve readability, we therefore introduce so-called *semantic updates* to capture the semantics of (syntactic) updates.

Definition 3.23. *Let* $(\text{VSym}, \text{FSym}_r, \text{FSym}_{nr}, \text{PSym}_r, \text{PSym}_{nr}, \alpha)$ *be a signature for a type hierarchy. A* semantic update *is a triple* $(f, (d_1, \ldots, d_n), d)$ *such that*

- $f : A_1, \ldots, A_n \to A \in \text{FSym}_{nr}$,
- $d_i \in \mathcal{D}^{A_i}$ $(1 \leq i \leq n)$, *and*
- $d \in \mathcal{D}^A$.

Since updates in general modify more than one location (a location is a pair $(f, (d_1, \ldots, d_n))$), we define sets of consistent semantic updates.

Definition 3.24. *A set CU of semantic updates is called* consistent *if for all* $(f, (d_1, \ldots, d_n), d), (f', (d_1', \ldots, d_m'), d') \in CU$,

$$d = d' \text{ if } f = f', n = m, \text{ and } d_i = d_i' \ (1 \leq i \leq n) \ .$$

Let $\mathcal{CU}$ *denote the set of consistent semantic updates.*

As we see in Def. 3.27, a syntactic update describes the modification of a state S as a set CU of consistent semantic updates. In order to obtain the state in which the terms, formulae, or updates following an update u are evaluated, CU is applied to S yielding a state S'.

Definition 3.25 (Application of semantic updates). *Let* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *be a signature for a given type hierarchy and let* $\mathcal{M} = (\mathcal{D}_0, \delta, \mathcal{I}_0)$ *be a first-order model for that signature.*

For any set $CU \in \mathcal{CU}$ *of consistent semantics updates, the* modification $CU(\mathcal{M})$ *is defined as the model* $(\mathcal{D}'_0, \delta', \mathcal{I}'_0)$ *with*

$$\begin{aligned} \mathcal{D}'_0 &= \mathcal{D}_0 \\ \delta' &= \delta \\ \mathcal{I}'_0(f)(d_1, \ldots, d_n) &= \begin{cases} d & \text{if } (f, (d_1, \ldots, d_n), d) \in CU \\ \mathcal{I}_0(f)(d_1, \ldots, d_n) & \text{otherwise} \end{cases} \end{aligned}$$

for all $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}_{nr}$ *and* $d_i \in \mathcal{D}^{A_i}$ $(1 \leq i \leq n)$.

Intuitively, a set CU of consistent semantic updates modifies the interpretation of $\mathcal{M}$ for the locations that are contained in CU.

Note 3.26. The consistency condition in Def. 3.24 guarantees that the interpretation function $\mathcal{I}'$ in Def. 3.25 is well-defined.

Definition 3.27 (Semantics of Java Card DL updates). *Given a signature for a type hierarchy, let* $\mathcal{K}_\preceq = (\mathcal{M}, \mathcal{S}, \rho)$ *be a Java Card DL Kripke structure with ordered domain, let* β *be a variable assignment, and let* $P \in \Pi$ *be a normalised Java Card program.*

For every state $S = (\mathcal{D}, \delta, \mathcal{I}) \in \mathcal{S}$*, the valuation function* val_S : *Updates* $\to$ $\mathcal{CU}$ *for updates is inductively defined by*

- $\mathrm{val}_{S,\beta}(f(t_1, \ldots, t_n) := s) = \{(f, (d_1, \ldots, d_n), d)\}$ *where*

$$\begin{aligned} d_i &= \mathrm{val}_{S,\beta}(t_i) \qquad (1 \leq i \leq n) \\ d &= \mathrm{val}_{S,\beta}(s) \ , \end{aligned}$$

- $\mathrm{val}_S\beta(u_1 \,;\, u_2) = (U_1 \cup U_2) \setminus C$ *where*

$$\begin{aligned} U_1 &= \mathrm{val}_{S,\beta}(u_1) \\ U_2 &= \mathrm{val}_{S',\beta}(u_2) \qquad \text{with } S' = \mathrm{val}_{S,\beta}(u_1)(S) \\ C &= \{(f, (d_1, \ldots, d_n), d) \mid (f, (d_1, \ldots, d_n), d) \in U_1 \text{ and} \\ &\qquad\qquad (f, (d_1, \ldots, d_n), d') \in U_2 \text{ for some } d' \neq d\} \ , \end{aligned}$$

- $\mathrm{val}_{S,\beta}(u_1 \,||\, u_2) = (U_1 \cup U_2) \setminus C$ *where*

$$\begin{aligned} U_1 &= \mathrm{val}_{S,\beta}(u_1) \\ U_2 &= \mathrm{val}_{S,\beta}(u_2) \\ C &= \{(f, (d_1, \ldots, d_n), d) \mid (f, (d_1, \ldots, d_n), d) \in U_1 \text{ and} \\ &\qquad\qquad (f, (d_1, \ldots, d_n), d') \in U_2 \text{ for some } d' \neq d\} \ , \end{aligned}$$

- $\mathrm{val}_{S,\beta}(\texttt{for}\ x;\ \phi;\ u) = U$ *where*

$$U = \{(f,(d_1,\ldots,d_n),d) \mid \textit{there is } a \in \mathcal{D}^A \textit{ such that } ((f,(d_1,\ldots,d_n),d),a) \in \textit{dom and } b \not\preceq a \textit{ for all } ((f,(d_1,\ldots,d_n),d'),b) \in \textit{dom}\}$$

with $\textit{dom} = \bigcup_{a\in\{d\in\mathcal{D}^A \mid S,\beta^d_x \models \phi\}} (\mathrm{val}_{S,\beta^a_x}(u) \times \{a\})$*, and* A *is the type of* x*,*
- $\mathrm{val}_{S,\beta}(\{u_1\}\ u_2) = \mathrm{val}_{S',\beta}(u_2)$ *with* $S' = \mathrm{val}_{S,\beta}(u_1)(S)$*.*

For an update u *without free variables we simply write* $\mathrm{val}_S(u)$ *since* $\mathrm{val}_{S,\beta}(u)$ *is independent of* β*.*

In both sequential and parallel updates, a later sub-update overrides an earlier one. The difference however is that with sequential updates the evaluation of the second sub-update is affected by the evaluation of the first one. This is not the case for parallel updates, which are evaluated simultaneously.

Example 3.28. Consider the updates

$$c := c+1\,;c := c+2$$

and

$$c := c+1\,||\,c := c+2$$

where c is a non-rigid constant. We stepwise evaluate these updates in a JAVA CARD DL state $S_1 = (\mathcal{D},\delta,\mathcal{I}_1)$ with $\mathcal{I}_1(c) = 0$.

$$\mathrm{val}_{S_1}(c := c+1\,;c := c+2) = (U_1 \cup U_2) \setminus C$$

where

$$\begin{aligned}
U_1 &= \mathrm{val}_{S_1,\beta}(c := c+1) = \{(c,(),1)\} \\
U_2 &= \mathrm{val}_{S_2,\beta}(c := c+2) = \{(c,(),3)\} \qquad \text{with } S_2 = \mathrm{val}_{S_1,\beta}(c := c+1)(S_1) \\
C &= \{(f,(d_1,\ldots,d_n),d) \mid (f,(d_1,\ldots,d_n),d) \in U_1 \text{ and } \\
&\qquad\qquad (f,(d_1,\ldots,d_n),d') \in U_2 \text{ for some } d' \neq d\} \\
&= \{(c,(),1)\}
\end{aligned}$$

That is, we first evaluate the sub-update $c := c+1$ in state S_1 yielding U_1. In order to evaluate the second sub-update, we first have to apply U_1 to state S_1, which results in the state S_2 that coincides with S_1 except for the interpretation of c, which is $\mathcal{I}_2(c) = 1$. The evaluation of $c := c+2$ in S_2 yields U_2. From the union $U_1 \cup U_2$ we have to remove the set C of conflicting semantic updates and finally obtain the result

$$\mathrm{val}_{S_1}(c := c+1\,;c := c+2) = \{(c,(),3)\}\ ,$$

i.e., a semantic update that fixes the interpretation of the 0-ary function symbol c to be the value 3.

On the other hand, the semantics of the parallel update in state S_1 is defined as

$$\mathrm{val}_{S_1}(c := c+1 \,||\, c := c+2) = (U_1 \cup U_2) \setminus C$$

where

$$\begin{aligned} U_1 &= \mathrm{val}_{S_1}(c := c+1) \\ U_2 &= \mathrm{val}_{S_1}(c := c+2) \\ C &= \{(f,(d_1,\ldots,d_n),d) \mid (f,(d_1,\ldots,d_n),d) \in U_1 \text{ and} \\ &\qquad\qquad (f,(d_1,\ldots,d_n),d') \in U_2 \text{ for some } d' \neq d\} \end{aligned}$$

That is, we first evaluate the two parallel sub-updates, resulting in the sets $U_1 = \{(c,(),1)\}$ and $U_2 = \{(c,(),2)\}$ of consistent semantic updates. Both U_1 and U_2 fix the interpretation of c for the same (and only) argument tuple $()$ but in an inconsistent way; U_1 and U_2 assign c the values 1 and 2, respectively. Such a situation is called a *clash*. As a consequence, the union $U_1 \cup U_2$ is not a set of consistent semantics updates. To regain consistency we have to remove those elements from the union that cause the clash. In the example, that is the set

$$C = \{(c,(),1)\} \ ,$$

and we obtain as the result

$$\mathrm{val}_{S_1}(c := c+1 \,||\, c := c+2) \ = \ \{(c,(),2)\} \ .$$

This example shows that in case of a clash within a parallel update, the later sub-update dominates the earlier one such that the evaluation of the second sub-update is not affected by the first one. In contrast, with sequential updates the first sub-update affects the second sub-update.

Not surprisingly, defining the semantics of quantified updates is rather complicated and proceeds in two steps.

First, we determine the set $D = \{d \in \mathcal{D}^A \mid S, \beta_x^d \models \phi\}$ of domain elements $d \in \mathcal{D}^A$ satisfying the guard formula ϕ with the variable assignment β_x^d. Then, for each $a \in D$ the sub-update u is evaluated with β_x^a resulting in a set of consistent semantic updates. If the quantified update is clash-free, its semantics is simply the union of all these sets of semantic updates.

In general though, a quantified update might contain clashes which must be resolved. For example, performing the steps described above for the quantified update

$$\texttt{for}\ x;\ x \doteq 0 \mid x \doteq 1;\ c := 5 - x$$

results in the two sets $U_1 = \{(c,(),5)\}$ and $U_2 = \{(c,(),4)\}$ of consistent semantic updates (the set of values satisfying the guard formula is $\{0,1\}$). The set $U_1 \cup U_2$ is inconsistent, i.e., the quantified update is not clash-free and we have to resolve the clashes. For this purpose it is important to remember the value $a \in D$ from the variable assignment β_x^a under which

the sub-update u was evaluated. Therefore, in Def. 3.27, we define a set $dom = \bigcup_{a \in \{d \in \mathcal{D}^A \mid S, \beta_x^d \models \phi\}} (\mathrm{val}_{S,\beta_x^a}(u) \times \{a\})$ consisting of pairs of (possibly inconsistent) semantics updates (resulting from $\mathrm{val}_{S,\beta_x^a}(u)$) and the appropriate value a (from β_x^a). In our example, the set dom is given as

$$dom = \{((c, (), 5), 0), ((c, (), 4), 1)\}$$

which we use for clash resolution. The clash is resolved by considering only one of the two semantic updates and discarding the other one. In general, to determine which one is kept the second components a, a' (here 0 and 1) from the elements in dom come into play: The semantic update $(f, (d_1, \ldots, d_n), d)$ with appropriate a is kept if $a \preceq a'$ for all $(f', (d'_1, \ldots, d'_n), d')$ with $f = f'$, $n = m$, $d_i = d'_i$ $(1 \leq i \leq n)$, and appropriate a'. That is, the semantic update arising from the least element satisfying the guard dominates. In our example we keep $(c, (), 5)$ and discard $(c, (), 4)$ since $0 \preceq 1$.

This "least element" approach for clash resolution in a sense carries over the last-win semantics of parallel updates to quantified updates. Note, that this is not the only possibility for clash resolution (see Note 3.30).

Example 3.29. In this example we show how clashes for quantified updates are resolved using the ordering predicate $\preceq$ on the domain.

Consider the update

$$\texttt{for}\ x;\ x \doteq 0 \mid x \doteq 1;\ h(0) := x \ .$$

It attempts to simultaneously assign the values 0 and 1 to the location $h(0)$. To keep the example simple, we assume that x ranges over the positive integers, which allows us to chose the usual "less than or equal" ordering relation $\leq$.

The semantics of a clashing quantified update is given by the least (second component of the) elements in the set dom with respect to the ordering. First, we determine the set dom (for an arbitrary state S):

$$\begin{aligned} dom &= \textstyle\bigcup_{a \,\in\, \{d \in \mathcal{D}^A \,\mid\, S, \beta_x^d \models (x \doteq 0 \mid x \doteq 1)\}} (\mathrm{val}_{S,\beta_x^a}(h(0) := x) \times \{a\}) \\ &= \{\mathrm{val}_{S,\beta_x^0}(h(0) := x) \times \{0\}, \mathrm{val}_{S,\beta_x^1}(h(0) := x) \times \{1\}\} \\ &= \{((h, (0), 0), 0), ((h, (0), 1), 1)\} \end{aligned}$$

since $\mathrm{val}_{S,\beta_x^0}(h(0) := x) = \{(h, (0), 0)\}$ and $\mathrm{val}_{S,\beta_x^1}(h(0) := x) = \{(h, (0), 1)\}$. Then, in the second step, we remove those elements from dom that cause clashes. In the example, the two elements are inconsistent, and we keep only $((h, (0), 0), 0)$ since it has the smaller second component with respect to the ordering $\leq$. That is, the result of evaluating the quantified update is the singleton set $\{(h, (0), 0)\}$ of consistent semantics updates.

As an example for a quantified update without clash consider

$$\texttt{for}\ x;\ x \doteq 0 \mid x \doteq 1;\ h(x) := 0 \ ,$$

which we evaluate in some arbitrary state S. Again we assume that x ranges over the non-negative integers. Then,

$$\begin{aligned} dom &= \bigcup_{a \in \{d \in \mathcal{D}^A \mid S, \beta_x^d \models x \doteq 0 | x \doteq 1\}} (\mathrm{val}_{S,\beta_x^a}(h(x) := 0) \times \{a\}) \\ &= \{\mathrm{val}_{S,\beta_x^0}(h(x) := 0) \times \{0\}, \mathrm{val}_{S,\beta_x^1}(h(x) := 0) \times \{1\}\} \\ &= \{((h,(0),0),0),((h,(1),0),1)\} \end{aligned}$$

This set *dom* does not contain inconsistencies and, thus, the semantics of the quantified update is $\{(h,(0),0),(h,(1),0)\}$.

Note 3.30. As already mentioned before there are several possibilities for defining the semantics of updates in case of a clash.

The crucial advantage of using a last-win clash semantics is that the transformation of sequential updates into parallel ones becomes almost trivial and can in practice be carried out very efficiently (⇒ Sect. 3.9.2). A last-win semantics allows to postpone case distinctions resulting from the possibility of aliasings/clashes to a later point in the proof.

Other possible strategies for handling clashes in quantified updates are (the basic ideas are mostly taken from the thesis by Platzer [2004b]):

- Leaving the semantics of updates undefined in case of a clash. This approach is similar to how partial functions (e.g., /) are handled in KeY. Then, a clashing update leads to a state where the location affected by the clash has a fixed but unknown value.
- Using the notion of *consistent* (syntactic) updates (as it is done in ASMs) in which no clashes occur. Following this idea, inconsistent updates would have no effect. However, according to the experiences with the existing version of KeY for ASMs [Nanchen et al., 2003], proving the consistency of updates tends to be tedious.
- Making the execution of updates containing clashes indeterministic (an arbitrary one of the clashing sub-updates is chosen). Then, however, updates would no longer be deterministic modal operators. Apart from the fact that the determinism of updates is utilised in a number of places in KeY, transformation rules for updates become much more involved for this clash semantics.

3.3.3 Semantics of Java Card DL Terms

The valuation function for Java Card DL terms is defined analogously to the one for first-order terms, though depending on the Java Card DL state.

Definition 3.31 (Semantics of Java Card DL terms). *Given a signature for a type hierarchy, let $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ be a KeY Java Card DL Kripke structure, let β be a variable assignment, and let $P \in \Pi$ be a normalised Java Card program.*

For every state $S = (\mathcal{D}, \delta, \mathcal{I}) \in \mathcal{S}$, the valuation function val_S *for terms is inductively defined by:*

$$\mathrm{val}_{S,\beta}(x) = \beta(x) \quad \textit{for variables } x$$

$$\mathrm{val}_{S,\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathrm{val}_{S,\beta}(t_1), \ldots, \mathrm{val}_{S,\beta}(t_n))$$

$$\mathrm{val}_{S,\beta}(\text{if } \phi \text{ then } t_1 \text{ else } t_2)) = \begin{cases} \mathrm{val}_{S,\beta}(t_1) & \textit{if } S, \beta \models \phi \\ \mathrm{val}_{S,\beta}(t_2) & \textit{if } S, \beta \not\models \phi \end{cases}$$

$$\mathrm{val}_{S,\beta}(\text{ifExMin } x.\phi \text{ then } t_1 \text{ else } t_2)) =$$

$$\begin{cases} \mathrm{val}_{S,\beta^d_x}(t_1) & \textit{if there is some } d \in \mathcal{D}^A \textit{ such that } S, \beta^d_x \models \phi \textit{ and} \\ & d \preceq d' \textit{ for any } d' \in \mathcal{D}^A \textit{ with } S, \beta^{d'}_x \models \phi \\ & \textit{(where } A \textit{ is the type of } x\textit{)} \\ \mathrm{val}_{S,\beta}(t_2) & \textit{otherwise} \end{cases}$$

$$\mathrm{val}_{S,\beta}(\{u\}(t)) = \mathrm{val}_{S_1,\beta}(t) \quad \textit{with } S_1 = \mathrm{val}_{S,\beta}(u)(S)$$

Since $\mathrm{val}_{S,\beta}(t)$ *does not depend on* β *if* t *is ground, we write* $\mathrm{val}_S(t)$ *in that case.*

The function and predicate symbols of a signature are divided into disjoint sets of rigid and non-rigid function and predicate symbols, respectively. From Def. 3.18 follows, that rigid symbols have the same meaning in all states of a given Kripke structure. The following syntactic criterion continues the notion of rigidness from function symbols to terms.

Definition 3.32. *A* Java Card *DL term* t *is* rigid

- *if* $t = x$ *and* $x \in \mathrm{VSym}$,
- *if* $t = f(t_1, \ldots, t_n)$, $f \in \mathrm{FSym}_r$ *and the sub-terms* t_i *are rigid* $(1 \leq i \leq n)$,
- *if* $t = \{u\}(s)$ *and* s *is rigid,*
- *if* $t = (\text{if } \phi \text{ then } t_1 \text{ else } t_2)$ *and the formula* ϕ *is rigid (Def. 3.35) and the sub-terms* t_1, t_2 *are rigid,*
- *if* $t = (\text{ifExMin } x.\phi \text{ then } t_1 \text{ else } t_2)$ *and the formula* ϕ *is rigid (Def. 3.35) and the sub-terms* t_1, t_2 *are rigid.*

Intuitively, rigid terms have the same meaning in all Java Card DL states (whereas the meaning of non-rigid terms may differ from state to state).

Lemma 3.33. *Let* $\mathcal{K}_\preceq = (\mathcal{M}, \mathcal{S}, \rho)$ *be a KeY Java Card DL Kripke structure, let* $P \in \Pi$ *be a normalised Java Card program, and let* β *be a variable assignment.*

If Java Card DL term t *is rigid, then*

$$\mathrm{val}_{S_1,\beta}(t) = \mathrm{val}_{S_2,\beta}(t)$$

for any two states $S_1, S_2 \in \mathcal{S}$.

The proof of the above lemma proceeds by induction on the term structure and makes use of the fact, that by definition the leading function symbol f of a term to be updated must be from the set FSym_{nr}.

3.3.4 Semantics of JAVA CARD DL Formulae

Definition 3.34 (Semantics of JAVA CARD DL formulae). *Given a signature for a type hierarchy, let $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ be a KeY JAVA CARD DL Kripke structure, let β be a variable assignment, and let $P \in \Pi$ be a normalised JAVA CARD program.*

For every state $S = (\mathcal{D}, \delta, \mathcal{I}) \in \mathcal{S}$ the validity relation $\models$ for JAVA CARD DL formulae is inductively defined by:

- $S, \beta \models p(t_1, \ldots, t_n)$ *iff* $(\mathrm{val}_{S,\beta}(t_1), \ldots, \mathrm{val}_{S,\beta}(t_n)) \in \mathcal{I}(p)$
- $S, \beta \models \mathrm{true}$
- $S, \beta \not\models \mathrm{false}$
- $S, \beta \models !\,\phi$ *iff* $S, \beta \not\models \phi$
- $S, \beta \models (\phi \,\&\, \psi)$ *iff* $S, \beta \models \phi$ *and* $S, \beta \models \psi$
- $S, \beta \models (\phi \mid \psi)$ *iff* $S, \beta \models \phi$ *or* $S, \beta \models \psi$ *(or both)*
- $S, \beta \models (\phi \texttt{->} \psi)$ *iff* $S, \beta \not\models \phi$ *or* $S, \beta \models \psi$ *(or both)*
- $S, \beta \models \forall x.\phi$ *iff* $S, \beta^d_x \models \phi$ *for every* $d \in \mathcal{D}^A$ *(where A is the type of x)*
- $S, \beta \models \exists x.\phi$ *iff* $S, \beta^d_x \models \phi$ *for some* $d \in \mathcal{D}^A$ *(where A is the type of x)*
- $S, \beta \models \{u\}(\phi)$ *iff* $S_1, \beta \models \phi$ *with* $S_1 = \mathrm{val}_{S,\beta}(u)(S)$
- $S, \beta \models \langle p \rangle \phi$ *iff there exists some state* $S' \in \mathcal{S}$ *such that* $(S, p, S') \in \rho$ *and* $S', \beta \models \phi$
- $S, \beta \models [p]\phi$ *iff* $S', \beta \models \phi$ *for every state* $S' \in \mathcal{S}$ *with* $(S, p, S') \in \rho$

We write $S \models \phi$ for a closed formula ϕ, since β is then irrelevant.

Similar to rigidness of terms, we now define rigidness of formulae.

Definition 3.35. *A JAVA CARD DL formula ϕ is* rigid

- *if* $\phi = p(t_1, \ldots, t_n)$, $p \in \mathrm{PSym}_r$ *and the terms t_i are rigid* $(1 \le i \le n)$,
- *if* $\phi = \mathrm{true}$ *or* $\phi = \mathrm{false}$,
- *if* $\phi = !\,\psi$ *and ψ is rigid,*
- $\phi = (\psi_1 \mid \psi_2)$, $\phi = (\psi_1 \,\&\, \psi_2)$, *or* $\phi = (\psi_1 \texttt{->} \psi_2)$, *and* ψ_1, ψ_2 *are rigid,*
- *if* $\phi = \forall x.\psi$ *or* $\phi = \exists x.\psi$, *and ψ is rigid,*
- $\phi = \{u\}\psi$ *and ψ is rigid.*

Note 3.36. A formula $\langle p \rangle \psi$ or $[p]\psi$ is *not* rigid, even if ψ is rigid, since the truth value of such formulas depends, e.g., on the termination behaviour of the program statements p in the modal operator. Whether a program terminates or not in general depends on the state the program is started in.

Intuitively, rigid formulae—in contrast to non-rigid formulae—have the same meaning in all JAVA CARD DL states.

Lemma 3.37. *Let $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ be a KeY JAVA CARD DL Kripke structure and let $P \in \Pi$ be a normalised JAVA CARD program, and let β be a variable assignment.*

If a JAVA CARD DL formula ϕ is rigid, then

$$S_1, \beta \models \phi \text{ if and only if } S_2, \beta \models \phi$$

for any two states $S_1, S_2 \in \mathcal{S}$.

Finally, we define what it means for a formula to be valid or satisfiable. A first-order formula is satisfiable (resp., valid) if it holds in some (all) model(s) for some (all) variable assignment(s) (⇒ Def. 2.40). Similarly, a JAVA CARD DL formula is satisfiable (resp. valid) if it holds in some (all) state(s) of some (all) Kripke structure(s) $\mathcal{K}_\preceq$ for some (all) variable assignment(s).

Definition 3.38. *Given a signature for a type hierarchy and a normalised JAVA CARD program* $P \in \Pi$*, let* ϕ *be a JAVA CARD DL formula.*

ϕ *is* satisfiable *if there is a KeY JAVA CARD DL Kripke structure* $\mathcal{K}_\preceq = (\mathcal{M}, \mathcal{S}, \rho)$ *such that* $S, \beta \models \phi$ *for some state* $S \in \mathcal{S}$ *and some variable assignment* β.

Given a KeY JAVA CARD DL Kripke structure $\mathcal{K}_\preceq = (\mathcal{M}, \mathcal{S}, \rho)$*, the formula* ϕ *is* $\mathcal{K}_\preceq$-valid*, denoted by* $\mathcal{K}_\preceq \models \phi$*, if* $S, \beta \models \phi$ *for all states* $S \in \mathcal{S}$ *and all variable assignments* β

ϕ *is* logically valid*, denoted by* $\models \phi$*, if* $\mathcal{K}_\preceq \models \phi$ *for all KeY JAVA CARD DL Kripke structures* $\mathcal{K}_\preceq$.

Note 3.39. Satisfiability and validity for JAVA CARD DL coincide with the corresponding notions in first-order logic (Def. 2.40), i.e., if a first-order formula ϕ is satisfiable (valid) in first-order logic then ϕ is satisfiable (valid) in JAVA CARD DL.

Note 3.40. The notions of satisfiability, $\mathcal{K}_\preceq$-validity, and logical validity of a formula depend on the given type hierarchy and normalised JAVA CARD program, and are not preserved if one of the two is modified—as the following simple example shows.

Suppose a type hierarchy contains an abstract type A with Null as its only subtype. Then the formula $\forall x.x \doteq \texttt{null}$, where x is of type A, is valid since $\mathcal{D}^A$ consists only of the element *null* to which the term `null` evaluates.

Now we modify the type hierarchy and add a dynamic type B that is a subtype of A and a supertype of Null. By definition, the domain of a dynamic type is non-empty, and, since B is a subtype of A, $\mathcal{D}^A$ contains at least one element $d \neq \mathit{null}$ with $d \in \mathcal{D}^B$. As a consequence, ϕ is not valid in the modified type hierarchy.

The following example shows that validity does not only depend on the given type hierarchy but also on the JAVA CARD DL program (which is, of course, more obvious). Suppose the type hierarchy contains the dynamic type *Base* with dynamic subtype *SubA*, and the normalised JAVA CARD program shown below is given:

Java

```
class Base {
  int m() {
    return 0;
  }

  public static void start(Base o) {
    int i=o.m();
  }
}

class SubA extends Base {
  int m() {
    return 0;
  }
}
```

Java

Consider the method invocation `o.m();` in line 7. Both the implementation of `m` in class `Base` and the one in class `SubA` may be chosen for execution. The choice depends on the dynamic type of the domain element that `o` evaluates to—resulting in a case distinction.

Nevertheless, the formula

$$\langle \texttt{i=Base.start(o);} \rangle \texttt{i} \doteq 0 \ ,$$

where $o{:}Base \in \mathrm{FSym}_{nr}$, is valid because both implementations of `m()` return 0.

Now we modify the implementation of method `m()` in class `SubA` by replacing the statement `return 0;` with `return 1;`. Then, ϕ is no longer valid since now there are values of `o` for which method invocation `o.m();` yields the return value 1 instead of 0.

Instead of modifying the implementation of method `m()` in class `SubA` one could also add another class `SubB` extending `Base` with an implementation of `m()` that return a "wrong" result, making ϕ invalid.

The examples given here clearly show that in general the validity of a formula depends on the given type hierarchy and on the program that is considered. As is explained in Sect. 8.5, as a consequence, any proof is in principle invalidated if the program is modified (e.g., by adding a new subclass) and needs to be redone. However, validity is not always lost if the program is modified and Sect. 8.5 presents methods for identifying situations where it is preserved.

Example 3.41. We now check the formulae from Example 3.15 for validity.

$\models \{c := 0\}\ (c \doteq 0)$ since in the state in which $c \doteq 0$ is evaluated, c is indeed 0 (due to the update).

$\not\models (\{c := 0\}\ c) \doteq c$ since $(\{c := 0\}c)$ evaluates to 0 in any state but there are states in which c (the right side) is different from 0.

$\models$ `sal` $\doteq$ `null` `->` $\langle$`ArrayList.demo2(sal);`$\rangle$`j` $\doteq 1$ since after invocation of `ArrayList.demo2(sal)` with an argument `sal` different from `null` the local variable `j` has the value 1.

$\not\models \{$`sal` $:=$ `null`$\}\langle$`ArrayList.demo2(sal);`$\rangle$`j` $\doteq 1$ since `j` has the value 0 when the program terminates if started in a state with `sal` $\doteq$ `null`. Due to the update `sal` $:=$ only such states are considered and, thus, this formula is even unsatisfiable.

$\not\models \{$`v` $:= g\}(\langle$`ArrayList al=new ArrayList(); v=al.inc(v);`$\rangle$`v` $\doteq g + 1)$ since the JAVA CARD addition `arg+1` in method `inc` causes a so-called *overflow* for $n = 2147483647$ ($\Rightarrow$ Chap. 12), in which case `v` has the negative value $-2147483648 \neq 2147483647 + 1$.

$\not\models \{$`v` $:= g\}(\langle$`ArrayList al=new ArrayList();`
`v=al.inc(v)@ArrayList`$\rangle$`v` $\doteq g + 1)$ This formula has the same semantics as the previous one since in this case the method-body statement `v=al.inc(v)@ArrayList` is equivalent to the method call `v=al.inc(v)` (because there are no subclasses of `ArrayList` overriding method `inc`).

$\models \langle$`int v=0;`$\rangle$`v` $\doteq 0$ since the program always terminates in state with `v` $\doteq 0$.

Example 3.42. The following two examples deal with functions that have a predefined fixed semantics (i.e., the same semantics in all Kripke seeds) only on parts of the domain. We consider the division function / (written infix in the following), which has a predefined interpretation for $\{(x, y) \in$ `integer` $\times$ `integer` $\mid y \neq 0\}$ while the interpretation for $\{(x, y) \in$ `integer` $\times$ `integer` $\mid y = 0\}$ depends on the particular Kripke seed.

$\mathcal{K}_{\preceq} \models 5/c \doteq 1$ in any $\mathcal{K}_{\preceq}$ with Kripke seed $\mathcal{M} = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$ such that
- $\mathcal{I}_0(c) = 5$ or
- $\mathcal{I}_0(c) = 0$ and $\mathcal{I}_0(5/0) = 1$.

$\models 5/c \doteq 5/c$ since $5/c \doteq 5/c$ holds in any state S of any $\mathcal{K}_{\preceq}$ (even if $\mathrm{val}_S(c) = 0$).

3.3.5 JAVA CARD-reachable States

As mentioned before, the set of states that are reachable by JAVA CARD programs is a subset of the states of a JAVA CARD DL Kripke structure. Indeed, a state is (only) JAVA CARD-reachable if it satisfies the following conditions:

1. A finite number of objects are created.[4]
2. Reference type attributes of non-null objects are either null or point to some other created object. Similarly, all entries of reference-type arrays different from null are either null or point to some created object.
3. For any array a the dynamic type $\delta(a[i])$ of the array entries is a subtype of the element type A of the dynamic type $A[\,] = \delta(a)$ of a (violating this condition in JAVA leads to an `ArrayStoreException`).

Given a type hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_a, \sqsubseteq)$, a signature, and a normalised JAVA CARD program $P \in \Pi$ the above conditions can be expressed with JAVA CARD DL formulae as follows (the implicit fields like, e.g., `<nextToCreate>` and the function T::get() used in the following formulae are defined formally in Sect. 3.6.6):

1. For all dynamic types $T \in \mathcal{T}_d \setminus \{\text{Null}\}$ with $T \sqsubseteq$ Object that occur in P:

$$\begin{aligned} &T.\texttt{<nextToCreate>} >= 0\ \&\\ &\forall x.(x >= 0\ \&\ x < T.\texttt{<nextToCreate>} <->\\ &\quad \text{T::get}(x).\texttt{<created>} \doteq \text{TRUE}) \end{aligned}$$

   where $x \in$ VSym is of type `integer`.
   By definition, an object o is created iff its index (which is the integer i for which the equation $o \doteq \text{C::get}(i)$ holds) is in the interval $[0, C.\texttt{<nextToCreate>}[$. This guarantees that only a finite number of objects are created. The above formula expresses the consistency between the implicit attribute $\text{T::get}(x).\texttt{<created>}$ and the definition of createdness for any type T.
2. (i) For all types $T \in \mathcal{T} \setminus \{\bot, \text{Null}\}$ with $T \sqsubseteq$ Object that occur in P and for all non-rigid function symbols $f : T \to T' \in \text{FSym}_{nr}$ with $T' \sqsubseteq$ Object that are declared as an attribute of T in P:

$$\begin{aligned} &\forall o.(o.\texttt{<created>} \doteq \text{TRUE}\ \&\ o\ !\doteq \mathit{null}) ->\\ &\quad (o.f \doteq \mathit{null} \mid o.f.\texttt{<created>} \doteq \text{TRUE})) \end{aligned}$$

   where $o \in$ VSym is of type T.

   (ii) For all array types $T[\,] \in \mathcal{T}$ that occur in P

$$\begin{aligned} &\forall a.\forall x.(a.\texttt{<created>} \doteq \text{TRUE}\ \&\ a\ !\doteq \mathit{null}) ->\\ &\quad (a[x] \doteq \mathit{null} \mid a[x].\texttt{<created>} \doteq \text{TRUE})) \end{aligned}$$

   where $a \in$ VSym is of type $T[\,]$ and $x \in$ VSym of type `integer`.
3. For all array types $T[\,] \in \mathcal{T}$ that occur in P:

$$\begin{aligned} &\forall a.\forall x.((a.\texttt{<created>} \doteq \text{TRUE}\ \&\ a\ !\doteq \mathit{null}) ->\\ &\quad \mathit{arrayStoreValid}(a, a[x])) \end{aligned}$$

[4] In JAVA CARD DL, objects are represented by domain elements, and the domain is assumed to be constant (see Note 3.19 on page 89). Whether an object is created or not is indicated by a Boolean function `<created>` (⇒ Sect. 3.6.6).

where $a \in$ VSym is of type $T[\,]$ and $x \in$ VSym of type `integer` (see App. A.2.2 for the semantics of the predicate *arrayStoreValid*).

Thus, there is a reachability formula for each type T, each reference type attribute, and each array type that occurs in the program P. Since the conjunction of all these may result in a quite lengthy formula, we introduce the non-rigid predicate *inReachableState* which, by definition, holds in a state S iff all the above formulae hold in S.

There are some more constraints restricting the set of JAVA CARD-reachable states dealing with class initialisation. For example, an initialised class is not erroneous. However, since class initialisation is not handled in this book we do not go into details here and omit the corresponding constraints (for a detailed account on class initialisation the reader is referred to [Bubel, 2001]). Please note that the KeY system can handle class initialisation.

Example 3.43. Consider the following JAVA CARD program

JAVA
```
class Control {
  Data data;
}

class Data {
  int d;
}
```
JAVA

We assume a specification of class `Data` that consists of the invariant

$$\forall data.(data.\texttt{<created>} \doteq \text{TRUE} \rightarrow data.\texttt{d} \geq 0)$$

(where $data \in$ VSym is of type `Data`), stating that, for all created objects of type `Data`, the value of the attribute `d` is non-negative.

Now, we would like to prove that for any object c of type `Control` that is created and different from `null`, the value of the attribute c`.data.d` is non-negative. With the semantics of JAVA CARD in mind, this seems to be a valid property given the invariant of class `Data`. However, the corresponding JAVA CARD DL formula

$$\begin{array}{l}\forall data.(data.\texttt{<created>} \doteq \text{TRUE} \rightarrow data.\texttt{d} \geq 0) \rightarrow \\ \quad (\texttt{c.<created>} \doteq \text{TRUE} \;\&\; \texttt{c} \mathrel{!\doteq} \texttt{null} \;\&\; \texttt{c.data} \mathrel{!\doteq} \texttt{null} \rightarrow \\ \qquad \texttt{c.data.d} \geq 0)\end{array}$$

is (surprisingly) not valid. The reason is that we cannot establish the assumption of the invariant of class `Data`, since we cannot prove that the equation `c.data.<created>` $\doteq$ TRUE holds, i.e., that `c.data` refers to a created object (even if we know that `c.data` is different from `null`). In JAVA CARD

it is always true that a non-null attribute of a created non-null object points to a created object. In our logic, however, we have to make this explicit by adding the assumption *inReachableState* stating that we are in a JAVA CARD-reachable state. We obtain

$$
\begin{array}{l}
(\mathit{inReachableState}\ \&\\
\forall \mathit{data}.(\mathit{data}.\texttt{<created>} \doteq \mathrm{TRUE} \rightarrow \mathit{data}.\texttt{d} \geq 0)) \rightarrow\\
\quad (\texttt{c.<created>} \doteq \mathrm{TRUE}\ \&\ \texttt{c} \mathrel{!\doteq} \texttt{null}\ \&\ \texttt{c.data} \mathrel{!\doteq} \texttt{null} \rightarrow\\
\qquad \texttt{c.data.d} \geq 0)
\end{array}
$$

One conjunct of *inReachableState* is the formula

$$
\begin{array}{l}
\forall o.(o.\texttt{<created>} \doteq \mathrm{TRUE}\ \&\ o \mathrel{!\doteq} \texttt{null}) \rightarrow\\
\quad (o.\texttt{data} \doteq \texttt{null} \mid o.\texttt{data.<created>} \doteq \mathrm{TRUE}))
\end{array}
$$

where $o \in \mathrm{VSym}$ is of type `Data`. If we instantiate the universal quantifier in this formula with `c` we can derive the desired equation

$$\texttt{c.data.<created>} \doteq \mathrm{TRUE}\quad .$$

This example shows that there are formulae that are true in all JAVA CARD-reachable states but that are not valid in JAVA CARD DL. This problem can be overcome by adding the predicate *inReachableState* to the invariants of the program to be verified. Then, states that are not reachable by any JAVA CARD program are excluded from consideration.

Dealing with the inReachableState Predicate in Proofs

When a correctness proof is started, the KeY system automatically adds the predicate *inReachableState* to the precondition of the specification. In the majority of cases, proofs can be completed without considering *inReachableState*. There are however situations that require the use of *inReachableState*:

(i) The proof can only be closed by employing (parts of) the properties stated in *inReachableState* (as in Example 3.43).
(ii) A state (described by some update) must be shown to satisfy the predicate *inReachableState*. Such a situation occurs, for example, when using a method contract (i.e., the specification of a method) in a proof. Then it is necessary to establish the precondition of the method specification, which usually contains the predicate *inReachableState*, in the invocation state (⇒ Sect. 3.8).

In both situations, expanding *inReachableState* into its components is not feasible since in practice the resulting formula would consist of hundreds or thousands of conjuncts.

To deal with situation (i), the KeY calculus provides rules that allow the user to extract parts of *inReachableState* that are necessary to close the proof.

To prevent full expansion of *inReachableState* in the case that

$$\mathit{inReachableState} \mathrel{-\!\!>} \{u\}\ \mathit{inReachableState}$$

must be shown for some update u (situation (ii)), the KeY system performs a syntactic analysis of the update u and expands only those parts of *inReachableState* that possibly are affected by u.

Note, that in general an update u that results from the symbolic execution of some program cannot describe a state that violates *inReachableState*. However, the user might provide such a malicious update that leads to an unreachable state by, for example, applying the cut rule (⇒ Sect. 3.5.2).

3.4 The Calculus for JAVA CARD DL

3.4.1 Sequents, Rules, and Proofs

The KeY system's calculus for JAVA CARD DL is a sequent calculus. It extends the first-order calculus from Chapter 2.

Sequents are defined as in the first-order case (Def. 2.42). The only difference is that now the formulae in the sequents are JAVA CARD DL formulae.

Definition 3.44. *A* sequent *is of the form $\Gamma \Longrightarrow \Delta$, where Γ, Δ are sets of closed JAVA CARD DL formulae.*

The left-hand side Γ is called antecedent *and the right-hand side Δ is called* succedent *of the sequent.*

As in the first-order case, the semantics of a sequent

$$\phi_1, \ldots, \phi_m \Longrightarrow \psi_1, \ldots, \psi_n$$

is the same as that of the formula

$$(\phi_1 \mathbin{\&} \ldots \mathbin{\&} \phi_m) \mathrel{-\!\!>} (\psi_1 \mid \ldots \mid \psi_m) \ .$$

In Chapter 2 we have used an informal notion of what a rule is, and what a rule application is. Now, we give a more formal definition.

Definition 3.45. *A* rule *R is a binary relation between (a) the set of all tuples of sequents and (b) the set of all sequents.*

If $R(\langle P_1, \ldots, P_k\rangle, C)$ ($k \geq 0$), then the conclusion *C is* derivable *from the* premisses *$P_1, \ldots, P_k$ using rule R.*

The set of sequents that are derivable *is the smallest set such that: If there is a rule in the (JAVA CARD DL) calculus that allows to derive a sequent S from premisses that are all derivable, then S is derivable in C.*

A *calculus*—in particular our JAVA CARD DL calculus—is formally a set of rules.

Proof trees are defined as in the first-order case (Def. 2.50), except that now the rules of the JAVA CARD DL calculus (as described in Sections 3.5–3.9) are used for derivation instead of the first-order rules. Intuitively, a proof for a sequent S is a derivation of S written as a tree with root S, where the sequent in each node is derivable from the sequents in its child nodes.

3.4.2 Soundness and Completeness of the Calculus

Soundness

The most important property of the JAVA CARD DL calculus is soundness, i.e., everything that is derivable is valid and only valid formulae are derivable.

Proposition 3.46 (Soundness). *If a sequent $\Gamma \Longrightarrow \Delta$ is derivable in the JAVA CARD DL calculus (Def. 3.45), then it is valid, i.e., the formula $\bigwedge \Gamma \texttt{->} \bigvee \Delta$ is logically valid (Def. 3.38).*

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 JAVA CARD DL calculus, the user can add new rules, whose soundness must then be proven w.r.t. the existing rules (⇒ Sect. 4.5).

Validating the Soundness of the JAVA CARD DL Calculus

So far, we have no intention of formally proving the soundness of the JAVA CARD DL 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 Sect. 4.5). Doing so would first require a formal specification of the JAVA CARD language. No *official* formal semantics of JAVA or JAVA CARD 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 favour. On the other hand, the KeY project performs ongoing cross-verification against other JAVA formalisations to ensure the faithfulness of the calculus.

One such effort compares the KeY calculus with the Bali semantics [von Oheimb, 2001a], which is a JAVA Hoare logic formalised 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 assignments.

Another validation was carried out by Ahrendt et al. [2005b]. A reference JAVA semantics from [Farzan et al., 2004] was used, which is formalised in Rewriting Logic [Meseguer and Rosu, 2004] and mechanised 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 frequent completely automated validation runs, which is beneficial, since the calculus is constantly extended with new features.

Furthermore, the KeY calculus is regularly tested against the compiler test suite Jacks (available at `www.sourceware.org/mauve/jacks.html`). 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.

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 JAVA CARD DL includes first-order arithmetic, which is already inherently incomplete as established by Gödel's Incompleteness Theorem [Gödel, 1931] (discussed in Sect. 2.7). Another, equivalent, argument is that a complete calculus for JAVA CARD DL would yield a decision procedure for the Halting Problem, which is well-known to be undecidable. Thus, a logic like JAVA CARD DL 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 formulae whose proof would require the derivation of a non-provable first-order property (being purely first-order, its provability would be independent of the program part of the calculus).

Proposition 3.47 (Relative Completeness). *If a sequent $\Gamma \Longrightarrow \Delta$ is valid, i.e., the formula $\bigwedge \Gamma \texttt{->} \bigvee \Delta$ is logically valid (Def. 3.38), then there is a finite set Γ_{FOL} of logically valid first-order formulae such that the sequent*

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

is derivable in the JAVA CARD DL 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 JAVA CARD DL formulae 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 JAVA CARD DL, is given by Platzer [2004b].

3.4.3 Rule Schemata and Schema Variables

The following definition makes use of the notion of *schema variables*. They represent concrete syntactical elements (e.g., terms, formulae or programs). Every schema variable is assigned a kind that determines which class of concrete elements is represented by such a schema variable.

Definition 3.48. *A* rule schema *is of the form*

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_k}{C} \qquad (k \geq 0)$$

where $P_1, \ldots, P_k$ *and* C *are schematic sequents, i.e., sequents containing schema variables.*

A rule schema $P_1 \cdots P_k \,/\, C$ represents a rule R if the following equivalence holds: a sequent C^* is derivable from premisses $P_1^*, \ldots, P_k^*$ iff $P_1^* \cdots P_k^* \,/\, C^*$ is an instance of the rule schema. Schema instances are constructed by instantiating the schema variables with syntactical constructs (terms, formulae, etc.) which are compliant to the kinds of the schema variables. One rule schema represents infinitely many rules, namely, its instances.

There are many cases, where a basic rule schema is not sufficient for describing a rule. Even if its general form adheres to a pattern that is describable in a schema, there may be details in a rule that cannot be expressed schematically. For example, in the rules for handling existential quantifiers, there is the restriction that (Skolem) constants introduced by a rule application must not already occur in the sequent. When a rule is described schematically, such constraints are added as a note to the schema.

All the rules of our calculus perform one (or more) of the following actions:

- A sequent is recognised as an axiom, and the corresponding proof branch is closed.
- A formula in a sequent is modified. A single formula (in the conclusion of the rule) is chosen to be in focus. It can be modified or deleted from the sequent. Note, that we do not allow more than one formula to be modified by a rule application.
- Formulae are added to a sequent. The number of formulae that are added is finite and is the same for all possible applications of the same rule schema.
- The proof branches. The number of new branches is the same for all possible applications of the same rule schema.

Moreover, whether a rule is applicable and what the result of the application is, depends on the presence of certain formulae in the conclusion.

The above list of possible actions excludes, for example, rules performing changes on all formulae in a sequent or that delete all formulae with a certain property.

Thus, all our rules preserve the "context" in a sequent, i.e., the formulae that are not in the focus of the rule remain unchanged. Therefore, we can simplify the notation of rule schemata, and leave this context out. Similarly, an update that is common to all premisses can be left out (this is formalised in Def. 3.49). Intuitively, if a rule "$\phi \Longrightarrow \psi \;/\; \phi' \Longrightarrow \psi'$" is correct, then $\phi' \Longrightarrow \psi'$ can be derived from $\phi \Longrightarrow \psi$ in all possible contexts. In particular, the rule then is correct in a context described by $\Gamma, \Delta, \mathcal{U}$, i.e., in all states s for which there is a state s_0 such that $\Gamma \Longrightarrow \Delta$ is true in s_0 and s is reached from s_0 by executing $\mathcal{U}$. Therefore, "$\Gamma, \mathcal{U}\phi \Longrightarrow \mathcal{U}\psi\ \Delta \;/\; \Gamma\,\mathcal{U}\phi' \Longrightarrow \mathcal{U}\psi',\ \Delta$" is a correct instance of "$\phi \Longrightarrow \psi \;/\; \phi' \Longrightarrow \psi'$", and $\Gamma, \Delta, \mathcal{U}$ do not have to be included in the schema. Instead we allow them to be added during application. Note, however, that the *same* $\Gamma, \Delta, \mathcal{U}$ have to be added to all premisses of a rule schema.

Later in the book (e.g., Sect. 3.7) we will present a few rules where the context cannot be omitted. Such rules are indicated with the $(*)$ symbol. These rules will be shown for comparison only; they are not part of the JAVA CARD DL calculus.

Definition 3.49. *If*

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

is an instance of a rule schema, then

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

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

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.

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 3.1 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 formulae and Γ, Δ to represent sets of formulae. Schema variables of corresponding kinds occur also in the *taclets* used to implement rules in the KeY system (⇒ Sect. 4.2).

Table 3.1. Correspondence between names of schema variables and their kinds

π	non-active prefix of JAVA code (Sect. 3.4.4)
ω	"rest" of JAVA code after the active statement (Sect. 3.4.4)
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 An access to an instance attribute $o\,.\,a$ is not simple because a `NullPointerException` may be thrown
nse	non-simple 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, ...	local program variables (i.e., non-rigid constants)
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 maches with 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 of the type of `j` is `int` (and not, e.g., `byte`).

3.4.4 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 non-active 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 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.

The postfix ω denotes the "rest" of the program, i.e., everything except the non-active 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 non-active 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 $\langle \pi pq\omega \rangle \phi$.

3.4.5 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 formulae. Since JAVA CARD DL formulae contain programs, the JAVA CARD DL 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 JAVA CARD DL 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

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

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 complex 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 analysing 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:

$$\texttt{o} \mathrel{!\doteq} \texttt{null} \Longrightarrow \{\texttt{v} := \texttt{o.next}\}\langle\texttt{v.prev=o;}\rangle\texttt{o.next.prev} \doteq \texttt{o}$$
$$\texttt{o} \doteq \texttt{null} \Longrightarrow \langle\texttt{throw new NullPointerException();}\rangle\texttt{o.next.prev} \doteq \texttt{o}$$

Thus, we can state the essence of symbolic execution: the Java code in the formulae 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 (⇒ Chap. 11) or loop invariants (⇒ Sect. 3.7.1). Also, certain kinds of loops can be translated into quantified updates [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.

3.4.6 Components of the Calculus

Our Java Card DL calculus has five major components, which are described in detail in the following sections. Since the calculus consists of hundreds of rules, however, we cannot list them all in this book. Instead, we give typical examples for the different rule types and classes (a complete list can be found on the KeY project website).

In particular, 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 loops (Sect. 3.7) and some of the rules for handling abrupt termination (Sect. 3.6.7).

The five components of the Java Card DL calculus are:

1. Non-program rules, i.e., rules that are not related to particular program constructs. This includes first-order rules, rules for data-types (in particular the integers), rules for modalities (e.g., rules for empty modalities), and the induction rule.
2. Rules that work towards reducing/simplifying the program and replacing it by a combination of case distinction (proof branches) and sequences of updates. These always (and only) apply to the first active statement. 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.
3. Rules that handle loops for which no fixed upper bound on the number of iterations exists. In this chapter, we only consider rules that handle loops using loop invariants (Sect. 3.7). A separate chapter is dedicated to handling loops by induction (Chapter 11).
4. Rules that replace a method invocation by the method's contract.
5. Update simplification.

Component 2 is the core for handling JAVA CARD programs occurring in formulae. 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 is to use the rules in Component 2, interspersed with applications of rules in Component 3 and Component 4 for handling loops resp. methods, to step-wise eliminate the program and replace it by updates and case distinctions. After each step, Component 5 is used to simplify/eliminate updates. The final result of this process are sequents containing pure first-order formulae. 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 problem that is 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 10).

3.5 Calculus Component 1: Non-program Rules

3.5.1 First-order Rules

Since first-order logic is part of JAVA CARD DL, all the rules of the first-order calculus introduced in Chapter 2 are also part of the JAVA CARD DL calculus. That is, all rules from Fig. 2.2 (classical first-order rules), Fig. 2.3 (equality rules), Fig. 2.4 (typing rules), and Fig. 2.5 (arithmetic rules) can be applied to JAVA CARD DL sequents—even if the formulae that they are applied to are not purely first-order.

Consider, for example, the rule impRight. In Chapter 2, the rule schema for this rule takes the following form:

$$\mathsf{impRight}\ \text{(Chapter 2 notation)}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \mathrel{->} \psi, \Delta}$$

In this chapter, we omit the context Γ, Δ from rule schemata (⇒ Sect. 3.4.3), i.e., the same rule is schematically written as:

$$\mathsf{impRight}\ \frac{\phi \Longrightarrow \psi}{\Longrightarrow \phi \mathrel{->} \psi}$$

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

$$\frac{x \doteq 1,\ \{x := 0\}(x \doteq y) \Longrightarrow \{x := 0\}\langle \texttt{m();}\rangle(y \doteq 0)}{x \doteq 1 \Longrightarrow \{x := 0\}(x \doteq y \mathrel{->} \langle \texttt{m();}\rangle(y \doteq 0))}$$

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

Due to the presence of modalities and non-rigid functions, which do not exist in purely first-order formulae, 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 needs to be restricted. That affects rules for universal quantification and equality rules.

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 $t \in \mathrm{Trm}_{A'}$ is a rigid ground term
whose type A' is a sub-type 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 JAVA CARD DL, however, we have to add the restriction that t is a *rigid* ground term (Def. 3.32). The reason is that, though an arbitrary value can be instantiated for x as it is universally quantified, in each individual instantiation, all occurrences of x must have the same value.

Example 3.50. The formula $\forall x.(x \doteq 0 \mathrel{->} \langle \texttt{i++;}\rangle(x \doteq 0))$ is logically valid, but instantiating the variable x with the non rigid constant `i` is wrong as it leads to the unsatisfiable formula $\texttt{i} \doteq 0 \mathrel{->} \langle \texttt{i++;}\rangle(\texttt{i} \doteq 0))$.

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

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

Restriction of Rules for Equalities

The equality rules (Fig. 2.3) are part of the JAVA CARD DL 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 (Def. 3.32), 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 Sect. 4.4.1.

Example 3.51. The sequent

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

is valid. But applying the equality to the occurrence of $\mathtt{v} + 1$ on the right side of the sequent is wrong, as it would lead to the sequent

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

that is satisfiable but *not valid*.

In the sequent

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

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 The Cut Rule and Lemma Introduction

The cut rule

$$\mathsf{cut}\ \frac{\Longrightarrow \phi \qquad \phi \Longrightarrow}{\Longrightarrow}$$

allows to introduce a lemma ϕ, which is an arbitrary JAVA CARD DL formula. The lemma occurs in the succedent of the left premiss (where, intuitively

speaking, the lemma has to be proved) and in the antecedent of the right premiss (where, intuitively speaking, the lemma can be used). One can also view the cut rule as a case distinction on whether ϕ is true or not as the right premiss is equivalent to $\Longrightarrow !\,\phi$.

Using the cut rule in the right way can greatly reduce the length of proofs. However, since the cut formula ϕ is arbitrary, the cut rule is not analytic and non-deterministic. That is the reason why it is not included in the calculus for first-order logic (it is not needed for completeness). In the KeY system it is only applied interactively when the user can choose a useful cut formula based on his or her knowledge and intuition.

The cut rule introduces a lemma ϕ that is proved in the particular context in which it is introduced. Thus, it can only be used in that context. It can, for example, not be used in the context of an update $\mathcal{U}$ since ϕ does not imply $\{\mathcal{U}\}\ \phi$. Another way to introduce a lemma is to define a new calculus rule and prove its soundness (⇒ Sect. 4.5). That way, a lemma ϕ can be introduced that can be used in any context (provided that ϕ is shown to be logically valid).

3.5.3 Non-program Rules for Modalities

The Java Card DL calculus contains some rules that apply to modal operators and are, thus, 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 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 Card.

3.6 Calculus Component 2: Reducing 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 `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 JAVA CARD DL calculus we use a different solution. It is based on the notion of *updates*, which can be seen as "semantic substitutions". Evaluating $\{loc := val\}\phi$ in a state is equivalent to evaluating ϕ in a modified state where *loc* evaluates to *val*, i.e., has been "semantically substituted" with *val* (see Sect. 3.2 for a discussion and a comparison of updates with assignments and substitutions).

The KeY system uses special simplification rules to compute the result of applying an update to terms and formulae that do not contain programs (⇒ Sect. 3.9). 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:

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

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

3.6.2 Rules for Handling General Assignments

There are four classes of rules in the JAVA CARD DL 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 in the left hand side, it cannot change the location scheduled for assignment, which is saved before the right-hand side is evaluated.

Step 1: Unfolding the Left-Hand Side

In this first step, the left-hand side of an assignment is unfolded if it is a non-simple 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 sub-expression is assigned.

If the left-hand side of the assignment is a non-atomic field access—which is to say it has the form $nse\,.\,a$, where nse is a non-simple expression—then the following rule is used:

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

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 resp. 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 analysed:

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

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

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

Step 2: Saving the Location

After the left-hand side has been unfolded completely (i.e., has the form v, $v\,.\,a$ or $v\texttt{[}se\texttt{]}$), the right-hand side has to be analysed. But before doing this, we have to memorise 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 memorising looks different for different kinds of expressions appearing on the left. The choice here is between field resp. array accesses. For local variables, the memorising 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 3.2:

$$\text{assignmentSaveLocation}\ \frac{\Longrightarrow \langle \pi\ \ \mathit{check}\texttt{;}\ \ \mathit{memorise}\texttt{;}\ \ \mathit{unfoldr}\texttt{;}\ \ \mathit{update}\texttt{;}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \ v\texttt{.}a\texttt{=}\mathit{nse}\texttt{;}\ \ \omega\rangle\phi}$$

Table 3.2. Components of rule assignmentSaveLocation for field accesses $v\texttt{.}a\texttt{=}\mathit{nse}$

check	`if (`v`==null) throw new NullPointerException();`	abort if v is `null`
memorise	$T_v\ \ v_0$ `=` v`;`	
unfoldr	$T_{nse}\ \ v_1$ `=` nse`;`	set up Step 3
update	v_0`.`a `=` v_1`;`	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 $v\texttt{[}\mathit{se}\texttt{]=}\mathit{nse}$. The components of the premiss for that case are shown in Table 3.3.

Table 3.3. Components of rule assignmentSaveLocation for array accesses $v\texttt{[}\mathit{se}\texttt{]=}\mathit{nse}$

check	`if (`se`>=`v`.length || `se`<0) throw new ArrayIndexOutOfBoundsException();`	abort if index out of bounds[a]
memorise	$T_v\ \ v_0$ `=` v`;` $T_{se}\ \ v_1$ `=` se`;`	
unfoldr	$T_{nse}\ \ v_2$ `=` nse`;`	set up Step 3
update	v_0`[`v_1`]` `=` v_2`;`	set up Step 4

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

Step 3: Unfolding the Right-Hand Side

In the next step, after the location that is changed by the assignment has been memorised, we can analyse 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 $nse\,.\,a$ with a non-simple object reference nse:

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

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

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 Java Card DL counterparts in order to obtain a proper logical term. For example, the Java division operator `/` has to be replaced by the function symbol *jdiv* (which is different from the standard mathematical division /, as explained in Chap. 12). The KeY system performs this conversion automatically to construct se^* from se. The complete list of predefined Java Card DL operators is given in App. A.

If there is an atomic field access $v\,.\,a$ 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:

$$\text{assignment}\ \frac{v \doteq null \Longrightarrow \{v_0 := v.a@Class\}\langle\pi\ \ \omega\rangle\phi \quad v = null \Longrightarrow \langle\pi\ \texttt{throw new NullPointerException();}\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ \ v_0\ \texttt{=}\ v\,.\,a\texttt{;}\ \ \omega\rangle\phi}$$

$$\text{assignment}\ \frac{v \doteq null \Longrightarrow \{v.a@Class := se^*\}\langle\pi\ \ \omega\rangle\phi \quad v = null \Longrightarrow \langle\pi\ \texttt{throw new NullPointerException();}\ \omega\rangle\phi}{\Longrightarrow \langle\pi\ \ v\,.\,a\ \texttt{=}\ se\texttt{;}\ \ \omega\rangle\phi}$$

A further complication is caused by field hiding. Hiding occurs when derived classes declare fields with the same name as in the superclass. The exact field reference has to be inferred from 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 disambiguated. This can be achieved by an up-front renaming (as proposed in Def. 3.10), or (as done in the KeY system) with on-the-fly disambiguation in the assignment rule. It adds a qualifier @*Class* to the name of the field as the field migrates from the program into the update. The pretty-printer does not display the qualifier if there is no hiding.

For array access, we have to consider the possibility of an `ArrayIndexOutOfBoundsException` 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):

$$\text{assignment}\quad \frac{\begin{array}{l} v \doteq null, se^* >= 0, se^* < v.length \Longrightarrow \{v_0 := v[se^*]\}\langle\pi\ \omega\rangle\phi \\ v = null \Longrightarrow \langle\pi\ \texttt{throw new NullPointerException();}\ \omega\rangle\phi \\ v \doteq null, (se^* < 0 \mid se^* >= v.length) \Longrightarrow \\ \quad \langle\pi\ \texttt{throw new ArrayIndexOutOfBoundsException();}\ \omega\rangle\phi \end{array}}{\Longrightarrow \langle\pi\ \ v_0\ \texttt{=}\ v\texttt{[}se\texttt{];}\ \omega\rangle\phi}$$

3.6.3 Rules for Conditionals

Most `if-else` statements have a non-simple 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

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

where v is a fresh boolean variable.

After dealing with the non-simple 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:

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

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 formula twice.

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

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

Note that the if-then-else in the premiss of the rule is a logical and not a program language construct (⇒ Def. 3.14).

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

$$(se \doteq \text{TRUE}) \mathrel{->} \langle \pi\ \ p\ \ \omega\rangle\phi \quad \& \quad (se\ !\doteq \text{TRUE}) \mathrel{->} \langle \pi\ \ q\ \ \omega\rangle\phi$$

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. 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. 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 bound, the loop has to be verified using (a) induction (⇒ Chap. 11) or (b) an invariant rule (⇒ Sect. 3.7.1, 3.7.4). 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:

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

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

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

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 behaviour: `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 JAVA CARD DL 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 Sect. 3.8). To obtain an efficient calculus we have conservatively extended the programming language (⇒ Def. 3.12) 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.

Evaluation of Method Invocation Expressions

The process of evaluating a method invocation expression (method call) within our JAVA CARD DL 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.

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.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 that this expression may point to. 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`.

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.

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.

Step 3: Evaluating the Arguments

If a method invocation has arguments, they have to be completely evaluated before control is transferred to the method body. This is achieved by the following rule:

methodCallUnfoldArguments

$$\frac{\begin{array}{l}\Longrightarrow \langle \pi \;\; T_{e_{l_1}} \;\; v_1\texttt{=}e_{l_1}\texttt{;} \\ \qquad\vdots \\ \qquad T_{e_{l_k}} \;\; v_k\texttt{=}e_{l_k}\texttt{;} \\ \qquad lhs \;\texttt{=}\; se\texttt{.}mname\texttt{(}a_1\texttt{,}\ldots\texttt{,}a_n\texttt{);} \\ \quad \omega\rangle\phi\end{array}}{\Longrightarrow \langle \pi \;\; lhs \;\texttt{=}\; se\texttt{.}mname\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{);} \;\; \omega\rangle\phi}$$

The rule unfolds the arguments using fresh variables in the usual manner. However, only those argument expressions e_i get unfolded that are non-simple. We refer to the non-simple argument expressions as $e_{l_1} \ldots e_{l_k}$. The rule only applies if $k > 0$, i.e., there is at least one non-simple argument expression. The expressions a_i used in the premiss of the rule are then defined by:

$$a_i = \begin{cases} e_i & \text{if } e_i \text{ is a simple expression} \\ v_i & \text{if } e_i \text{ is a non-simple expression} \end{cases}$$

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.

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

$$\frac{se \doteq \texttt{null} \Longrightarrow \langle \pi \ \texttt{throw new NullPointerException();} \ \omega \rangle \phi \qquad se \mathbin{!\doteq} \texttt{null} \Longrightarrow \langle \pi \ T_{lhs} \ v_0\texttt{;} \ paramDecl\texttt{;} \ ifCascade\texttt{;} \ lhs \ \texttt{=} \ v_0\texttt{;} \ \omega \rangle \phi}{\Longrightarrow \langle \pi \ lhs \ \texttt{=} \ se\texttt{.}mname\texttt{(}se_1\texttt{,}\ldots\texttt{,}se_n\texttt{);} \ \omega \rangle \phi}$$

The code piece *paramDecl* introduces and initialises 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 and refer to the distinct method implementations via `method-body` statements (⇒ Def. 3.12). Altogether, *ifCascade* abbreviates:

$$\begin{array}{l}
\texttt{if (}se\ \texttt{instanceof}\ C_1\texttt{)} \\
\quad v_0\texttt{=}se\texttt{.}mname\texttt{(}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{)@}C_1\texttt{;} \\
\texttt{else if (}se\ \texttt{instanceof}\ C_2\texttt{)} \\
\quad v_0\texttt{=}se\texttt{.}mname\texttt{(}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{)@}C_2\texttt{;} \\
\vdots \\
\texttt{else if (}se\ \texttt{instanceof}\ C_{k-1}\texttt{)} \\
\quad v_0\texttt{=}se\texttt{.}mname\texttt{(}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{)@}C_{k-1}\texttt{;} \\
\texttt{else}\ v_0\texttt{=}se\texttt{.}mname\texttt{(}p_1\texttt{,}\ldots\texttt{,}p_n\texttt{)@}C_k\texttt{;}
\end{array}$$

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 $mname(\ldots)$. In other words, the more specialised 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.

Step 5: Creating the Method Frame

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

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

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

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:

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

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.

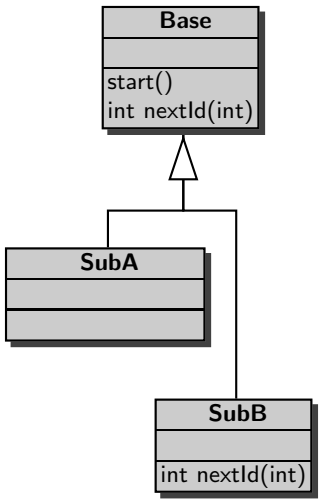
After the assignment of the return value is symbolically executed, we are left with an empty method frame, which can now be removed altogether. This is achieved with the rule

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

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.

Example for Handling a Method Invocation

Consider the example program from Fig. 3.3. 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




```
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;
  }
}
```

Fig. 3.3. An example program with method overriding

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.

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; {
    int i_1 = i;
    if (o instanceof SubB)
        j=o.nextId(i_1)@SubB;
    else
        j=o.nextId(i_1)@Base;
}
res=j;
```
(A)
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 a method body statement pointing to the implementation from `SubB`:

JAVA
```
j=o.nextId(i_1)@SubB;
res=j;
```
(A′)
JAVA

Now 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=SubB, this=o) : {
    return super.nextId(i_1)+1;
}
res=j;
```
(B) (A″)
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=SubB, this=o) : {
    int j_2 = super.nextId(i_1);
    j_1=j_2+1;
    return  j_1;
}
res=j;
```
(C) (B′)
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=SubB, this=o) : {
    int j_3; {
        int i_2 = i_1;
        j_3=o.nextId(i_2)@Base;
    }
    j_2=j_3;
     j_1=j_2+1;
     return  j_1;
}
res=j;
```

(C′)

Java

Now it is necessary to remove the declarations and perform the assignments to reach the method body statement `j_3=o.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:

Java

```
method-frame(result->j, source=SubB, this=o) : {
    method-frame(result->j_3, source=Base, this=o) : {
        return ++i_2;
    }
    j_2=j_3;
     j_1=j_2+1;
     return j_1;
}
res=j;
```

(D) (C″)

Java

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:

JAVA

```
method-frame(result->j, source=SubB, this=o) : {
    method-frame(result->j_3, source=Base, this=o) : {
        return j_4;
    }
     j_2=j_3;
     j_1=j_2+1;
     return j_1;
}
res=j;
```

(D′) (E)

JAVA

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=SubB, this=o) : {
    method-frame(source=Base, this=o) : {
        j_3=j_4;
    }
     j_2=j_3;
     j_1=j_2+1;
     return j_1;
}
res=j;
```

(E′)

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=SubB, this=o) : {
    method-frame(source=Base, this=o) : {
    }
     j_2=j_3;
     j_1=j_2+1;
     return j_1;
}
res=j;
```

(E″)

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=SubB, this=o) : {
    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 Initialisation

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

Instance Creation and the Constant Domain Assumption

Java Card DL, 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 *object repositories* with access functions and *implicit fields* that allow to change and query the program-visible instance state (created, initialised, etc.). 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) attributes, their names are enclosed in angled brackets. According to their use we distinguish object state and repository fields. An overview of the used implicit fields is given in Table 3.4.

Definition 3.52. *Given a non-abstract class type C, the object repository Rep_C is the set of all domain elements e of dynamic type C:*

$$Rep_C := \{e \in \mathcal{D}_0 \mid \delta(e) = C\}$$

Note, that Rep_C does not contain the objects of type D even if D is a subtype of C.

Allocating a new object requires to access the corresponding object repository. Therefore, we introduce access functions for the object repositories.

Definition 3.53. *For each non-abstract class type C there is a predefined rigid function symbol*

$$\mathrm{C}\mathit{::get} : \texttt{integer} \to C$$

called repository access function.

Restricted to the set of non-negative integers, C*::get is interpreted as a bijective mapping onto the object repository Rep_C of type C. For negative integers,* C*::get is also defined, but its values are unknown.*

Given a Java Card *DL Kripke structure, the* index *of an object o is the non-negative integer i for which the equation $\mathcal{I}(\mathrm{C}\mathit{::get})(i) = o$ holds.*

Example 3.54. Since the dynamic type function $\delta(\cdot)$ is only defined for a model, it cannot be used within the logic. We must take another way to express with a formula that a term ("expression") evaluates ("refers") to a domain element ("object") of a given dynamic type. The repository access functions allow us to do it concisely. For example the formula

$$\exists i : \texttt{integer}.(o \doteq \mathrm{C}{::}\mathrm{get}(i))$$

holds iff the term o evaluates to a domain element of dynamic type C (excluding, among other, elements of any type D, which might be a subtype of C).

To model instance allocation appropriately, we must ensure that the new object is not already in use. Therefore, we declare an implicit static integer field `<nextToCreate>` for each non-abstract class type C.

We call an object *created*, if its index is greater or equal to zero and less than the value of `<nextToCreate>`. When an instance of dynamic type T is allocated by a Java program, the instance with T`.<nextToCreate>` as object index is used and `<nextToCreate>` is incremented by one. In all states that are reachable by a Java program (⇒ Sect. 3.3.5), the value of `<nextToCreate>` is non-negative.

Table 3.4. Implicit object repository and status fields

Modifier	Implicit field	Declared in	Explanation
`private static`	`int <nextToCreate>`	T	the index of the object to be taken the next time when a `new` T`(...)` expression is evaluated
`protected`	`boolean <created>`	`Object`	indicates whether the object has been created
`protected`	`boolean <initialised>`	`Object`	indicates whether the object has been initialised

Further, there is the implicit boolean instance field `<created>` declared in `java.lang.Object`, which is supported mainly for convenience. This field is set for an object during the instance creation process.

Example 3.55. Consider an instance invariant of class `A` (a property that must hold for every object of class `A` or any class derived from `A`, in each observable state) that states that the field `head` declared in `A` must always be non-null.

With `<created>` this can be formalised concisely as:

$$\forall a : A.(a\texttt{.<created>} \doteq \text{TRUE} \mathrel{-\!\!>} (a\texttt{.head} \mathrel{!\!\doteq} \texttt{null}))$$

Using `<nextToCreate>` and the repository access functions, in contrast, would yield a complicated formula, and even require enumerating all subtypes of `A` in it.

The Java Instance Initialisation Process

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

The transformation inserts code for explicitly executing all initialisation 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 3.5.

Table 3.5. Implicit methods for object creations and initialisation declared in every non-abstract type T (syntactic conventions from Figure 3.4)

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 initialisers and the invoked constructor

The transformation covers all details of initialisation in Java, except that we only consider non-concurrent programs and no reflection facilities (in particular no instances of `java.lang.Class`). Initialisation of classes and interfaces (also known as static initialisation) is fully supported for the single

threaded case. KeY passes the static initialisation challenge stated by Jacobs et al. [2003]. We omit the treatment of this topic here for space reasons though. In the following we use the schematic class form that is stated in Figure 3.4.

```
mods0 class T {
  mods1 T1 a1 = initExpression1 ;
  ⋮
  modsm Tm am = initExpressionm ;

  {
    initStatementm+1 ;
    ⋮
    initStatementl ;
  }

  mods T(params) {
     st1 ;
     ⋮
     stn ;
  }
  ...
}
```

Fig. 3.4. Initialisation part in a schematic class

Example 3.56. Figure 3.5 shows a class `Person` and its mapping to the schematic class declaration of Figure 3.4. There is only one initialiser 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`

Fig. 3.5. Example for the mapping of a class declaration to the schema of Fig. 3.4

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.

The Rule for Instance Creation and Initialisation

The instance creation rule

$$\textsf{instanceCreation}\ \frac{\begin{array}{l}\Longrightarrow \langle \pi\ T\ v_0\ \texttt{=}\ T\texttt{.<createObject>();} \\ \qquad v_0\texttt{.<init>(}args\texttt{);} \\ \qquad v_0\texttt{.<initialised> = true;} \\ \qquad v\ \texttt{=}\ v_0\texttt{;} \\ \omega\rangle\phi\end{array}}{\Longrightarrow \langle \pi\ v\ \texttt{=}\ \texttt{new}\ T\texttt{(}args\texttt{);}\ \omega\rangle\phi}$$

replaces an instance creation expression "v `=` `new` T`(`$args$`)`" by a sequence of statements. These can be divided into three phases, which we examine in detail below:

1. obtain the next available object from the repository (as explained above, it is not really "created") and assign it to a fresh temporary variable v_0
2. prepare the object by assigning all fields their default values
3. initialise the object of v_0 and subsequently mark it as initialised. Finally, assign v_0 to v

The reason for assigning to v in the last step is to ensure correct behaviour in case initialisation terminates abruptly due to an exception.[5]

Phase 1: Instance Creation

The implicit static method `<createObject>()` (⇒ Fig. 3.6) declared in each non-abstract class T returns the next available object from the object repository of type T after setting its fields to default values.

`<createObject>()` delegates the actual interaction with the object repository to yet another helper, an implicit method called `<allocate>()`. The `<allocate>()` method has no Java implementation, its semantics is given by the following rule instead:

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

```
public static T <createObject>() {
        // Get an unused instance from the object repository
    T newObject = T.<allocate>();
    newObject.<transient> = 0;
    newObject.<initialized> = false;
        // Invoke the preparation method to assign default values to
        // instance fields
    newObject.<prepare>();
        // Return the newly created object in order to initialise it:
    return newObject;
}
```

Fig. 3.6. Implicit method `<createObject>()`

allocateInstance

$$\frac{\Longrightarrow \{lhs := T :: get(T.\texttt{<nextToCreate>})\}\ \{lhs.\texttt{<created>} := true\}\ \{T.\texttt{<nextToCreate>} := T.\texttt{<nextToCreate>} + 1\}\langle \pi\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ lhs\ \texttt{=}\ T.\texttt{<allocate>();}\ \omega\rangle\phi}$$

The rule ensures that after termination of `<allocate>()`:

- The object that has index `<nextToCreate>` (in the pre-state) is allocated and returned.
- Its `<created>` field has been set to true.
- The field `<nextToCreate>` has been increased by one.

Note that the mathematical arithmetic addition is used to specify the increment of field `<nextToCreate>`. This is the reason for using a calculus rule to define `<allocate>()` instead of JAVA code. An unbounded number of objects could not be modelled with bounded integer data types of JAVA.

Phase 2: Preparation

The next phase during the execution of `<createObject>()` is the preparation phase. All fields, including the ones declared in the superclasses, are assigned their default values.[6] 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., 2000, § 4.5.5]. The method `<prepare>()` used for preparation is shown in Figure 3.7.

[6] 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.

```
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);
}
```

Fig. 3.7. Implicit method `<prepare>()`

Note 3.57. In the KeY system, `<createObject>()` 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.

Phase 3: Initialisation

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

- as a field initialiser expression "*T attr* `=` *val*`;`" (e.g., `(*)` in Figure 3.8); the corresponding initialiser statement is *attr* `=` *val*;
- as an instance initialiser block (similar to `(**)` in Figure 3.8); such a block is also an initialiser statement;
- within a constructor body (like `(***)` in Figure 3.8).

For each constructor *mods T*`(`*params*`)` of *T* we provide a constructor normal form *mods T* `<init>(`*params*`)`, which includes (1) the initialisation of the superclass, (2) the execution of all initialiser statements in source code order, and finally (3) the actual constructor body. In the initialisation 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 3.8.

The exact blueprint for building a constructor normal form is shown in Figure 3.9, using the conventions of Figure 3.4. 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 re-start of the initialisation phase for the superclass of *T*. If $T =$ `Object` stop. Afterwards, initialiser statements

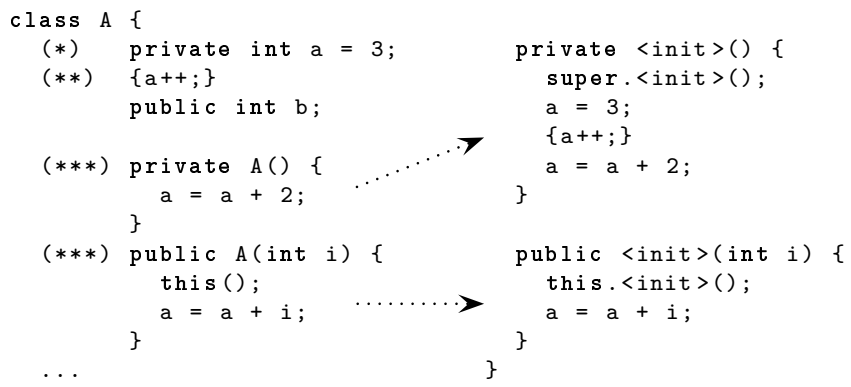


Fig. 3.8. Example for constructor normal forms

are executed in source code order. Finally, the original constructor body is executed.

2. st_1 = `this(`*args*`)`: Recursive re-start of the initialisation phase with the alternate constructor. Afterwards, the original constructor body is executed.

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

3.6.7 Handling Abrupt Termination

Abrupt Termination in Java Card DL

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 loop or a `switch` statement 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).

Evaluation of Arguments

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

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

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

    // add the initialiser
    // 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 initialiser 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

Fig. 3.9. Building the constructor normal form

If the Whole Program Terminates Abruptly

In Java Card DL, 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 *non-terminating* statement (Def. 3.18). For that case rules such as the following are provided in the Java Card DL calculus for all abruptly terminating statements:

$$\text{throwDiamond}\ \frac{\Longrightarrow \text{false}}{\Longrightarrow \langle \texttt{throw}\ se;\ \omega \rangle \phi} \qquad \text{throwBox}\ \frac{\Longrightarrow \text{true}}{\Longrightarrow [\texttt{throw}\ se;\ \omega] \phi}$$

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.

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 JAVA CARD DL 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 Sect. 3.6.5).
- 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 (Sect. 3.7).

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

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):

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

Rules for `try` and `throw`

The rule in Figure 3.10 allows to handle `try-catch-finally` blocks and the `throw` statement. 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 p 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:

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

Fig. 3.10. The rule for `try-catch-finally` and `throw`

$$\text{tryFinallyThrow}\ \frac{\Longrightarrow \langle \pi\ \ T_{se}\ \ v_{se}\ \texttt{=}\ se\texttt{;}\ \ r\ \texttt{throw}\ v_{se}\texttt{;}\ \ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{try \{ throw }se\texttt{; }p\texttt{ \} finally \{ }r\texttt{ \}}\rangle\phi}$$

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:

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

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` resp. the `return` statement (a different abruptly terminating statement that may occur in the `finally` block takes precedence). This behaviour is simulated by the following two rules (here, also, cs is a finite, possibly empty sequence of catch clauses):

tryBreak

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

tryReturn

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

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 resp. by a break statement with a label l identical to one of the labels $l_1, \ldots, l_k$ of the block ($k \geq 1$).

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

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

To blocks (labelled or unlabelled) 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:

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

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

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

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

3.7 Calculus Component 3: Invariant Rules for Loops

There are two techniques for handling loops in KeY: induction and using an invariant rule. In the following we describe the use of invariant rules. A separate chapter is dedicated to handling loops by induction (Chapter 11).

3.7.1 The Classical Invariant Rule

Before we discuss the problems that arise when setting up an invariant rule for a complex language like JAVA CARD, we first recall the classical invariant rule for a simple deterministic while-language with assignments, if-then-else, and while-loops. In particular, we assume that there is no abrupt termination and expressions do not have side-effects.

For such a simple while-language the invariant rule looks as follows:

$$\text{invRuleClassical}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U} Inv, \ \Delta \\ Inv, \ se \Longrightarrow [p] Inv \\ Inv, \ ! \, se \Longrightarrow \phi \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\texttt{while (}se\texttt{) \{ } \ p \ \texttt{\}}]\phi, \ \Delta} \ (*)$$

This rule states that, if one can find a formula *Inv* such that the three premisses hold requiring that

(a) *Inv* holds in the beginning,
(b) *Inv* is indeed an invariant, and
(c) the conclusion ϕ follows from *Inv* and the negated loop condition ! *se*,

then ϕ holds after executing the loop (provided it terminates). Remember that the symbol $(*)$ in the rule schema means, that the context $\Gamma, \Delta, \mathcal{U}$ must be empty unless its presence is stated explicitly (as in the first premiss), i.e., only instances of the schema itself are inference rules.

It is crucial to the soundness of Rule invRuleClassical that expressions are free of side-effects and that there is no concept of abrupt termination like, for example, in Java Card. In the following we discuss the problems arising from side-effects of expressions and abrupt termination concerning the invariant rule.

3.7.2 Loop Invariants and Abrupt Termination in Java Card DL

Java Card DL does not distinguish non-termination and abrupt termination. For example, the formulae

$$[\texttt{while (true) ;}]\phi$$

and

$$[\texttt{i = i / (j - j);}]\phi$$

are equivalent (both evaluate to true). However, the program (fragment) in the first formula does not terminate while the program in the second formula terminates abruptly with an `ArithmeticException` (due to division by zero).

Thus, setting up a sound invariant rule for Java Card DL requires a more fine-grained semantics concerning termination behaviour of programs. There are (at least) the following two approaches to distinguish between non-termination and abrupt termination.

Firstly, the logic Java Card DL could be enriched with additional labelled modalities $[\,]_R$ and $\langle\,\rangle_R$ with $R \subseteq \{\mathit{break}, \mathit{exception}, \mathit{continue}, \mathit{return}\}$ referring to the reason R of a possible abrupt termination. The semantics of a formula $[p]_R\phi$ is that, if the program p terminates abruptly with reason R, then the formula ϕ has to hold in the final state, whereas $\langle p\rangle_R\phi$ expresses that p terminates abruptly with reason R and in the final state ϕ holds.

The second possibility for distinguishing non-termination and abrupt termination is to perform a program transformation such that the resulting program catches all top-level exceptions and thus always terminates normally. Abrupt termination due to exceptions can, e.g., be handled by enclosing the original program with a `try-catch` block. For example, the following (valid) formula expresses that if the program from above terminates abruptly with an exception then formula ϕ has to hold:

$$[\texttt{Throwable thrown = null;}\\ \texttt{try \{}\\ \texttt{\quad i = i / (j - j);}\\ \texttt{\} catch (Exception e) \{}\\ \texttt{\quad thrown = e;}\\ \texttt{\}}\\ ](\texttt{thrown} \doteq \texttt{null} \rightarrow \phi)$$

Using the additional modalities the same could be expressed more concisely as

$$[\texttt{i = i / (j - j);}]_{exception}\phi \ .$$

Handling the other reasons for abrupt termination by program transformation is more involved and is not explained here. The advantage of using dedicated modalities is that termination properties can be addressed on a syntactic level (suited for reasoning with a calculus), whereas the program transformation approach relies on the semantics of JAVA CARD to encode abrupt termination and its reason. In order to describe the invariant rule for JAVA CARD DL in the following (and also the improved rule in Section 3.7.4), we pursue the approach of introducing additional modalities. Note however, that the actual rule available in the KeY system is based on the program transformation approach. The reason for that is that introducing indexed modalities would result in a multitude of rules to be added to the calculus.

Since the general rule covering all reasons for abrupt termination and side-effects of the loop condition is very complex, we start with some simpler rules dealing with special cases excluding certain difficulties (e.g., abrupt termination).

Normal Termination and Condition without Side-effects

Under the assumptions that (a) the loop does not terminate abruptly (i.e., terminates normally or does not terminate at all) and that (b) the loop condition does not have side-effects, the invariant rule essentially corresponds to the classical rule (the only difference are the prefix π and the postfix ω, which are not present in the classical rule. As a reminder: π consists of opening braces, labels, and try-statement but no executable statements and ω contains the rest of the program (including closing braces and catch-blocks).

$$\mathsf{invRuleSimple}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\,Inv,\ \Delta\\ Inv\ \&\ se \Longrightarrow [p]Inv\\ Inv\ \&\ !\,se \Longrightarrow [\pi\ \omega]\phi\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{while (}se\texttt{) \{ }p\texttt{ \} }\omega]\phi,\ \Delta}\ (*)$$

Abrupt Termination and Condition without Side-effects

When a `continue` statement without label or with a label referring to the currently investigated loop is encountered in the loop body, the execution of

the body is stopped and the loop condition is evaluated again, i.e., the loop moves on to the next iteration. Thus, the invariant *Inv* has to hold both when the body terminates normally and when a continue statement occurs (second premiss of rule invRuleAt).

If during the execution of the loop body

- a `continue` statement with a label referring to an enclosing loop,
- an exception is thrown that is not caught within the body,
- a `break` statement occurs without label or with a label that refers to a position in front of the loop, or
- a `return` statement occurs

then the whole loop statement terminates abruptly. In the rule, the reasons leading to abrupt termination of the whole loop statement are contained in the set $AT = \{break, exception, return\}$. Note that a `continue` with a label referring to an enclosing loop can be simulated by a corresponding `break` statement and, thus, we also use the label *break* to identify this case. In contrast, we use the label *continue* if the control flow is to be transferred to the beginning of the loop containing the `continue` statement.

The consequence of abrupt termination of a loop statement is that the execution of the loop is stopped immediately and the control flow is changed according to the reason for the abrupt termination. Thus, since the whole loop statement terminates, it is then not necessary to show that the invariant holds. Rather it must be shown that the postcondition ϕ holds after the rest of the program following the while loop has been executed (provided it terminates). This is expressed by the third premiss of rule invRuleAt, where the formula $\langle p \rangle_{AT}$ true holds iff p terminates abruptly (se is a simple expression and its evaluation to terminate normally). If no abrupt termination occurs in the loop body, rule invRuleAt reduces to rule invRuleSimple).

$$\text{invRuleAt}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\,Inv,\ \Delta \\ Inv\ \&\ se \Longrightarrow [p]Inv\ \&\ [p]_{continue}Inv \\ Inv\ \&\ se \Longrightarrow \langle p \rangle_{AT}\,\text{true} \texttt{ -> } [\pi\ \ p\ \ \omega]\phi \\ Inv\ \&\ !\,se \Longrightarrow [\pi\ \omega]\phi\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{while (} se \texttt{) \{ } p \texttt{ \} } \omega]\phi,\ \Delta}\ (*)$$

Normal Termination and Condition with Side-effects

Now we assume that the loop body terminates normally but the loop condition may have side-effects (including abrupt termination which leads to abrupt termination of the loop statement). A loop condition *nse* that may have side-effects is not a logical term and therefore has to be evaluated first. The result is then assigned to a new variable v of type `boolean`. The only reason for abrupt termination of a while statement during evaluation of the loop condition can be an exception. Thus, $AT = \{exception\}$.

The first premiss is identical to the one in the previous rules. The second and third premiss correspond to premisses two and three of rule invRuleSimple, but take possible side-effects (except for abrupt termination) of evaluating the loop condition e into account.

Abrupt termination of the loop condition caused by an exception is handled in premiss four, where the postcondition ϕ has to be established since the whole loop statement terminates abruptly. If the evaluation of the loop condition does not throw an exception this premiss trivially holds since then $\langle \texttt{v=}e\texttt{;}\rangle_{AT}\,\text{true}$ evaluates to false.

In case that the evaluation of *nse* does not terminate at all, all premisses except for the first one are trivially valid. Note that this would not be true if modality $[\cdot]_{AT}$ had been used in the fourth premiss instead of $\langle\cdot\rangle_{AT}$.

$$\text{invRuleNse}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv},\ \Delta\\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{TRUE} \texttt{ -> } [p]\mathit{Inv})\\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{FALSE} \texttt{ -> } [\pi\ \omega]\phi)\\ \mathit{Inv},\ \langle\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}\rangle_{AT}\,\text{true} \Longrightarrow [\pi\ \texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}\ \omega]\phi\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{while (}\mathit{nse}\texttt{) \{}\ p\ \texttt{\}}\ \omega]\phi,\ \Delta}\ (*)$$

Abrupt Termination and Condition with Side-effects

The following most general rule covers all possible cases of side-effects and abrupt termination.

Again, the sets $AT = \{\mathit{break}, \mathit{exception}, \mathit{return}\}$ and $AT' = \{\mathit{exception}\}$ contain the reasons leading to abrupt termination of the whole loop statement.

invRule

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\mathit{Inv},\ \Delta\\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{TRUE} \texttt{ -> } ([p]\mathit{Inv}\ \&\ [p]_{\mathit{continue}}\mathit{Inv}))\\ \mathit{Inv},\ \langle\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}\rangle_{AT'}\,\text{true} \Longrightarrow [\pi\ \texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}\ \omega]\phi\\ \mathit{Inv},\ \langle\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}\rangle(v \doteq \text{TRUE}\ \&\ \langle p\rangle_{AT}\,\text{true}) \Longrightarrow\\ \qquad\qquad [\pi\ \texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}\,p\ \omega]\phi\\ \mathit{Inv} \Longrightarrow [\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{FALSE} \texttt{ -> } [\pi\ \omega]\phi)\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \texttt{while (}\mathit{nse}\texttt{) \{}\ p\ \texttt{\}}\ \omega]\phi,\ \Delta}\ (*)$$

The first premiss is identical to the one in previous rules and states that the invariant has to hold in the beginning.

The second premiss covers the case that executing the loop body once preserves the invariant both if the execution terminates normally and if a `continue` statement occurred. Note that it is important that the execution of p is started in a state reflecting the side-effects of evaluating the loop condition *nse*. Therefore, writing

$$\mathit{Inv}, [\texttt{boolean}\ \ v\texttt{=}\mathit{nse}\texttt{;}](v \doteq \text{TRUE}) \Longrightarrow [p]\mathit{Inv}\ \&\ [p]_{\mathit{continue}}\mathit{Inv}$$

instead would not be correct.

Premiss three (which is the same as premiss four of rule invRuleNse) states that if the invariant holds and the evaluation of the loop condition *nse* terminates abruptly (the only reason can be an exception), then the postcondition ϕ has to hold after the rest ω of the program has been executed. Since the whole loop terminates abruptly if the loop condition terminates abruptly, the loop is omitted in the formula on the right side of the sequent.

Also in the fourth premiss, the postcondition ϕ has to be established if the invariant *Inv* holds and the evaluation of the loop condition *nse* terminates normally but the execution of p terminates abruptly due to one of the reasons in AT. Again, in the formula on the right side of the sequent the loop statement is replaced by a statement evaluating the loop condition and the body of the loop.

The last premiss applies to the case that the loop terminates because the loop condition evaluates to false. Then, assuming the invariant *Inv* holds before executing the rest ω of the program, the postcondition ϕ has to hold.

3.7.3 Implementation of Invariant Rules

The invariant rules, from invRuleSimple to invRule, are different from other rules of the JAVA CARD DL calculus, since in some premisses the context (denoted by Γ, Δ and the update $\mathcal{U}$) is deleted, which is why rule schemata with $(*)$ are used ($\Rightarrow$ Def. 3.49). If the context would not be deleted, the rules would not be sound as the following example shows.

Example 3.58. Consider the following JAVA CARD program:

JAVA
```
while ( i<10 ) {
  i=i+1;
}
```
JAVA

The sequent

$$\texttt{i} \doteq 0 \Longrightarrow [\texttt{int i=0; while (i<10) \{ i=i+1; \}}]\texttt{i} \doteq 0$$

is obviously *not* valid. In our example program no abrupt termination can occur and the loop condition has no side-effects. We thus can apply the rule invRuleSimple. Let us see what happens, if we use the following (unsound) variant where the context is not deleted from the second and the third premiss (schema version without $(*)$):

$$\textsf{unsoundRule}\ \frac{\begin{array}{l}\Longrightarrow \mathit{Inv}\\ \mathit{Inv}\ \&\ \mathit{se} \Longrightarrow [p]\mathit{Inv}\\ \mathit{Inv}\ \&\ !\,\mathit{se} \Longrightarrow [\pi\ \omega]\phi\end{array}}{\Longrightarrow [\pi\ \texttt{while (}\mathit{se}\texttt{) \{ }p\texttt{ \} }\omega]\phi}$$

Instantiating that rule yields (with $\Gamma = (\texttt{i} \doteq 0)$, Δ,$\mathcal{U}$ empty, and using the formula true as invariant):

$$\text{unsoundInstance}\ \frac{\begin{array}{l}\texttt{i} \doteq 0 \Longrightarrow \text{true} \\ \texttt{i} \doteq 0,\ \text{true}\ \&\ \texttt{i} < 10 \Longrightarrow [\texttt{i=i+1;}]\text{true} \\ \texttt{i} \doteq 0,\ \text{true}\ \&\ !(\texttt{i} < 10) \Longrightarrow []\texttt{i} \doteq 0\end{array}}{\texttt{i} \doteq 0 \Longrightarrow [\texttt{int i=0; while (i<10) \{ i=i+1; \}}]\texttt{i} \doteq 0}$$

As one can easily see, the three premisses of this instance are valid but the conclusion is not. The reason for this unsoundness is that the context describes the initial state of the loop execution; however, for correctness, in the second premiss the loop body would have to be executed in an arbitrary state (that is described merely by the invariant) and in the third premiss the invariant and the negated loop condition must entail the postcondition in the final state of the loop execution.

If the invariant rule is to be implemented using the taclet language presented in Chapter 4, there is the problem that taclets do not allow to omit context formulae since they act locally on the formula or term in focus. This is a deliberate design decision and not a flaw of the taclet language, which in most cases is very useful. However, in the case of the invariant rule, it requires to use some additional mechanism. The implementation of the invariant rule using taclets is based on the idea that a special kind of updates can be used to achieve a similar effect as with omitting the context. These special updates are called *anonymising* updates and their intuitive semantics is that they assign arbitrary unknown values to all locations, thus "destroying" the information contained in the context.

Definition 3.59 (Anonymising Update). *Let a* Java Card *DL signature* $(\text{VSym}, \text{FSym}_r, \text{FSym}_{nr}, \text{PSym}_r, \text{PSym}_{nr}, \alpha)$ *for a type hierarchy, a normalised* Java Card *program* $P \in \Pi$*, and a sequent* $\Gamma \Longrightarrow \Delta$ *be given.*

For every $f_i : A_1, \ldots, A_{n_i} \to A \in \text{FSym}_{nr}$ $(0 \leq i \leq n)$ *occurring in* P *or in* $\Gamma \cup \Delta$ *let* $f'_i : A_1, \ldots, A_{n_i} \to A \in \text{FSym}_r$ *be a fresh (w.r.t.* P *and* $\Gamma \cup \Delta$*) rigid function symbol (i.e.,* f'_i *does neither occur in* P *nor in* $\Gamma \cup \Delta$*). Then, the update*

$$u_1 \,||\, u_2 \,||\, \cdots \,||\, u_n$$

with

$$u_i \;=\; \texttt{for}\ x^i_1;\ \text{true};\ \cdots \texttt{for}\ x^i_{n_i};\ \text{true};\ f_i(x^i_1, \ldots, x^i_{n_i}) := f'_i(x^i_1, \ldots, x^i_{n_i})$$

is called an anonymising update *for the sequent* $\Gamma \Longrightarrow \Delta$*. In the following we abbreviate an anonymising update with* $\mathcal{V}$*.*

In the KeY system, the syntax for anonymising updates is $\{* := *n\}$, where $n \in \mathbb{N}$.

Using anonymising updates we can set-up invariant rules for Java Card DL that can be implemented using the taclet language. Here, we only present the

variant of the general rule invRule, covering abrupt termination and side-effects of the loop condition (variants of the rules invRuleSimple and invRuleNse with anonymising updates are obtained analogously).

invRuleAnonymisingUpdate

$$\frac{\begin{array}{l}\Longrightarrow Inv\\ \mathcal{V}Inv \Longrightarrow \mathcal{V}[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{TRUE} \texttt{ -> } ([p]Inv \,\&\, [p]_{continue}Inv))\\ \mathcal{V}Inv,\ \mathcal{V}\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle_{AT'}\text{true} \Longrightarrow \mathcal{V}[\pi\ v\texttt{=}nse\texttt{;}\ \omega]\phi\\ \mathcal{V}Inv,\ \mathcal{V}\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle(v \doteq \text{TRUE} \,\&\, \langle p\rangle_{AT}\text{true}) \Longrightarrow \mathcal{V}[\pi\ v\texttt{=}nse\texttt{;}p\ \omega]\phi\\ \mathcal{V}Inv \Longrightarrow \mathcal{V}[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{FALSE} \texttt{ -> } [\pi\ \omega]\phi)\end{array}}{\Longrightarrow [\pi\ \texttt{while (}nse\texttt{) \{ }p\texttt{ \}}\ \omega]\phi}$$

As can be seen, the context remains unchanged in all premisses, but formulae whose evaluation must not be affected by the context are prefixed with an anonymising update $\mathcal{V}$.

Note 3.60. Please note that the above rule looks differently in the KeY system since in the implementation we follow the approach based on a program transformation to deal with abrupt termination of the loop instead of introducing additional modalities $\langle\cdot\rangle_{AT}$ and $[\cdot]_{AT}$ (see the discussion in Sect. 3.7.2).

3.7.4 An Improved Loop Invariant Rule

Performance and usability of program verification systems can be greatly enhanced if specifications of programs and program parts not only consist of the usual pre-/postcondition pairs and invariants but also include additional information, such as knowledge about which memory locations are changed by executing a piece of code. More precisely, we associate with a (sequence of) statement(s) p a set Mod_p of expressions, called the modifier set (for p), with the understanding that Mod_p is part of the specification of p. Its semantics is that those parts of a program state that are *not* referenced by an expression in Mod_p are never changed by executing p [Beckert and Schmitt, 2003].

Usually, modifier sets are used for method specifications (⇒ Chap. 5, 8). In this chapter we extend the idea of modifier sets to loops. Similar as with method specifications, modifier sets for loops allow to

- separate the aspects of (a) which locations change and (b) how they change,
- state the change information in a compact way,
- make the proof process more efficient.

To achieve the latter point, we define a new JAVA CARD DL proof rule for while loops that makes use of the information contained in a modifier set for the loop. The main idea is to throw away only those parts of the context Γ, Δ and $\mathcal{U}$ (i.e., of the descriptions of the initial state) that may be changed by the loop. Anything that remains unchanged is kept and can be used to

establish the invariant (second premiss of rule invRuleAnonymisingUpdate) and the postcondition (premisses 3–5 of rule invRuleAnonymisingUpdate). That is, we do not use an anonymising update $\mathcal{V}$ as in rule invRuleAnonymisingUpdate which assigns unknown values to *all* location but a more restricted update that only assigns anonymous values to *critical* locations.

An important advantage of using modifier sets is that usually a loop only changes few locations, and that only these locations need to be put in a modifier set. On the other hand, using the traditional rule, all locations that are *not* changed and whose value is of relevance have to be included in the invariant and, typically, the number of relevant locations that are not changed by a loop is much bigger than the number of locations that are changed. Of course, in general, not everything that remains unchanged is needed to establish the postcondition in the third premiss. But when applying the invariant rule it is often not obvious what information must be preserved, in particular if the loop is followed by a non-trivial program. That can lead to repeated failed attempts to construct the right invariant. Whereas, to figure out the locations that are (possibly) changed by the loop, it is usually enough to look at the small piece of code in the loop condition and the loop body.

As a motivating example, consider the following JAVA CARD program fragment $p_{\min}$ that computes the minimum of an array `a` of integers:

JAVA (3.1)

```
m = a[0]; i = 1;
while (i < a.length) {
  if (a[i] < m) then
    m = a[i];
  i++;
}
```

JAVA

A postcondition (in KeY syntax) for this program is

$$\phi_{\min} = \forall x.(0 <= x \;\&\; x < \texttt{a.length} \rightarrow \texttt{m} <= \texttt{a[}x\texttt{]}) \;\&\; \exists x.(0 <= x \;\&\; x < \texttt{a.length} \;\&\; \texttt{m} \doteq \texttt{a[}x\texttt{]}) \;,$$

stating that, after running $p_{\min}$, the variable `m` indeed contains the minimum of `a`. However, a specification that just consists of $\phi_{\min}$ is rather weak. The problem is that $\phi_{\min}$ can also be established using, for example, a program that sets `m` as well as all elements of `a` to 0, which of course is not the *intended* behaviour. To exclude such programs, the specification must also state what the program does modify (the variables `i` and `m`) and does not modify (the array `a` and its elements). One way of doing this is to extend the postcondition with an additional part

$$\phi_{\text{inv}} = \forall x.(0 <= x \;\&\; x < \texttt{a.length} \rightarrow \texttt{a[}x\texttt{]} \doteq \texttt{a\_old[}x\texttt{]})$$

where `a_old` is a new array variable (not allowed to occur in the program) that is supposed to contain the "old" values of the array elements. To make sure `a_old` has the same elements as `a`, the formula ϕ_{inv} must also be used as a precondition and, thus, be turned into an invariant. In JAVA CARD DL, this specification of $p_{\min}$ is written as $\phi_{\text{inv}} \texttt{ -> } [p_{\min}](\phi_{\min} \texttt{ \& } \phi_{\text{inv}})$.

In order to prove the correctness of the program $p_{\min}$ using the classical invariant rule invRule or the variant with anonymising updates (rule invRuleAnonymisingUpdate), it is crucial to add the formula ϕ_{inv} also to the loop invariant that is used. Otherwise, the loop invariant is not strong enough to entail the postcondition ϕ (the third premiss of the loop rule does not hold). The reason is that all premisses of the invariant rule except for the first one omit the context formulae Γ, Δ and the sequence $\mathcal{U}$ of updates, i.e., all information about the state reached before running the while loop is lost (one can construct similar examples where the second premiss of the rule does not hold). The only way to keep this information—as long as no modifier sets are used—is to add it to the invariant. In general, loop invariants are "polluted" with formulae stating what the loop does *not* do. All relevant properties of the pre-state that need to be preserved have to be encoded into the invariant, even if they are in no way affected by the loop. Thus, two aspects are intermingled:

- Information about what intended effects the loop *does* have.
- Information about what non-intended effects the loop *does not* have.

This problem can be avoided by encoding the second aspect (i.e., the change information) with a modifier set instead of adding it to the invariant. Then the correctness of the program and the correctness of the modifier set can be shown in independent proofs and, thus, the two aspects are separated on verification level as well.

Modifier Sets

We shall now formally define the notion of modifier sets which has been motivated above. Intuitively, a modifier set enumerates the set of locations that a code piece p may change—it is thus part of the specification of p. The question how correctness of a modifier set with respect to a program can be proved is addressed in Sect. 8.3.3.

In general programs and in particular loops can—and in practice often do—change a finite but *unknown* number of locations (though in our simple motivating example $p_{\min}$ the number of changed locations is known to be two). A loop may, for example, change all elements in a list whose length is not known at proof time but only at run time. Therefore, to handle loops, we define modifier sets that can describe location sets of unknown size. Of course, such modifier sets can no longer be represented as simple enumerations of ground terms. Rather, we use guard formulae to define the set of ground terms

that may change (this is similar to the use of guard formulae in quantified updates).

Definition 3.61 (Syntax of Modifier Sets). *Let* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *be a signature for a type hierarchy.*

A modifier set *Mod is a set of pairs* $\langle \phi, f(t_1, \ldots, t_n)\rangle$ *with* $\phi \in$ *Formulae and* $f(t_1, \ldots, t_n) \in$ *Terms with* $f \in \mathrm{FSym}_{nr}$.

Given a sequent $\Gamma \Longrightarrow \Delta$, *let* $F \subseteq \mathrm{FSym}_{nr}$ *be the set of non-rigid function symbols* $f \in \mathrm{FSym}_{nr}$ *occurring in* $\Gamma \cup \Delta$. *Then,* $\{*\}$ *is the modifier set*

$$\bigcup_{f \in F} \{\langle \mathrm{true}, f(x_1, \ldots, x_n)\rangle\}$$

(specifying that any location in $\Gamma \Longrightarrow \Delta$ *may change).*

The intuitive meaning of a modifier set is that some location $(f, (d_1, \ldots, d_n))$ may be changed by a program p when started in a state S, if the modifier set for p contains an element $\langle \phi, f(t_1, \ldots, t_n)\rangle$ and there is variable assignment β such that the following conditions hold:

1. $\mathrm{val}_{S,\beta}(t_i) = d_i$ for $1 \leq i \leq n$, i.e., β assigns the free logical variables occurring in t_i values such that t_i coincides with d_i.
2. $S, \beta \models \phi$, i.e., the guard formula ϕ holds for the variable assignment β.

For our example program $p_{\min}$, an appropriate modifier set is

$$\mathit{Mod}_{\min} = \{\langle \mathrm{true}, \mathtt{i}\rangle, \langle \mathrm{true}, \mathtt{m}\rangle\} \ .$$

It states in a very compact and simple way that $p_{\min}$ only changes `i` and `m` and, in particular, does *not* change the array `a`.

A modifier set *Mod* is said to be correct for a program p if p (at most) changes the value of locations mentioned in *Mod*.

Definition 3.62 (Semantics of Modifier Sets). *Given a signature for a type hierarchy, let* $\mathcal{K}_{\preceq} = (\mathcal{M}, \mathcal{S}, \rho)$ *be a KeY JAVA CARD DL Kripke structure, and let* β *be a variable assignment.*

A pair $(S_1, S_2) = ((\mathcal{D}, \delta, \mathcal{I}_1), (\mathcal{D}, \delta, \mathcal{I}_2)) \in \mathcal{S} \times \mathcal{S}$ *of states* satisfies *a modifier set Mod, denoted by*

$$(S_1, S_2) \models \mathit{Mod} \ ,$$

iff, for

(a) all $f : A_1, \ldots, A_n \to A \in \mathrm{FSym}_{nr}$,
(b) all $(d_1, \ldots, d_n) \in \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$

the following holds:

$$\mathcal{I}_1(f)(d_1, \ldots, d_n) \neq \mathcal{I}_2(f)(d_1, \ldots, d_n)$$

implies that there is a pair $\langle \phi, f(t_1, \ldots, t_n) \rangle \in Mod$ *and a variable assignment* β *such that*

$$d_i = \mathrm{val}_{S_1,\beta}(t_i) \qquad (1 \leq i \leq n)$$

and

$$S_1, \beta \models \phi \ .$$

The modifier set Mod is correct *for a program p, if*

$$(S_1, S_2) \models Mod$$

for all state pairs $(S_1, p, S_2) \in \rho$.

The definition states that, if after running a program p there is some location $(f, (d_1, \ldots, d_n))$ which is assigned a value different from its value in the pre-state, then the modifier set must contain a corresponding entry, i.e., a pair $\langle \phi, f(t_1, \ldots, t_n) \rangle$ such that for some variable assignment β the formula ϕ holds and the t_i evaluate to d_i $(1 \leq i \leq n)$. Note that ϕ and the t_i are evaluated in the pre-state.

Example 3.63. Consider the following JAVA CARD method, which has one parameter of type `int[]`.

JAVA

```
public multiplyByTwo(int[] a) {
  int i = 0;
  int j = 0;
  while (i < a.length) {
    a[i] = a[i] * 2;
    i++;
  }
}
```

JAVA

Since `a` is a parameter of the method, the value of `a.length` is unknown. Thus, for giving a correct modifier set, it is not possible to enumerate the locations `a[0]`, `a[1]`, ..., `a[a.length-1]`.

However, a correct modifier set for the above program can be written as

$$\{\langle 0 <= x \ \& \ x < \texttt{a.length}, \texttt{a}[x] \rangle, \langle \mathrm{true}, \texttt{i} \rangle\} \ .$$

Another correct modifier set illustrating that modifier sets are not necessarily minimal is

$$\{\langle 0 <= x \ \& \ x < \texttt{a.length}, \texttt{a}[x] \rangle, \langle \mathrm{true}, \texttt{i} \rangle, \langle \mathrm{true}, \texttt{j} \rangle\} \ .$$

In general, a correct modifier set describes a superset of the locations that actually change.

The modifier set $\{\langle 0 <= x \ \& \ x < \texttt{a.length}, \texttt{a}[x] \rangle\}$ is not correct for the above program, since `i` is changed by the program but not contained in the modifier set.

Based on a modifier set *Mod*, we define the notion of an *anonymising update with respect to Mod*, which is an update that (only) assigns unknown values to those locations that are contained in the modifier set *Mod*.

Definition 3.64 (Anonymising Update w.r.t. a Modifier Set). *Let a signature* $(\mathrm{VSym}, \mathrm{FSym}_r, \mathrm{FSym}_{nr}, \mathrm{PSym}_r, \mathrm{PSym}_{nr}, \alpha)$ *for a type hierarchy, a modifier set Mod, and a sequent* $\Gamma \Longrightarrow \Delta$ *be given. For every* $\langle \phi_i, f_i(t_1^i, \ldots, t_{n_i}^i)\rangle \in Mod$ *with* $f_i : A_1, \ldots, A_{n_i} \to A$*, let* $f_i' \in \mathrm{FSym}_r$ *be a fresh (w.r.t.* $\Gamma \cup \Delta$*) rigid function symbol with the same type as* f_i*, i.e.,* f_i' *does not occur in* $\Gamma \cup \Delta$*.*

Then the update $\mathcal{V}(Mod) =$

$$\begin{cases} \mathcal{V} & \text{if } Mod = \{*\} \\ u_1 \,||\, \cdots \,||\, u_k & \text{if } Mod = \{\langle \phi_1, f_1(t_1^1, \ldots, t_{n_1}^1)\rangle, \ldots, \langle \phi_k, f_k(t_1^k, \ldots, t_{n_k}^k)\rangle\} \end{cases}$$

with

$\mathcal{V}$ *being an anonymising update for the sequent* $\Gamma \Longrightarrow \Delta$ *(Def. 3.59) and* $u_i = \texttt{for}\ x_1^i;\ \mathrm{true};\ \cdots \texttt{for}\ x_{l_i};\ \phi_i;\ f_i(t_1^i, \ldots, t_{n_i}^i) := f_i'(t_1^i, \ldots, t_{n_i}^i)$

where

$$\{x_1^i, \ldots, x_{l_i}^i\} = fv(\phi_i) \cup fv(t_1^i) \cup \cdots \cup fv(t_{n_i}^i)$$

is called an anonymising update with respect to *Mod.*

Properties of Anonymising Updates w.r.t. *inReachableState*

An anonymising update $\mathcal{V}(Mod)$ assigns terms an unknown but fixed value. As a consequence, the state it describes is not necessarily reachable by a JAVA CARD program. The idea of an anonymising update however is that it approximates all possible state changes of some program and, thus, we require that anonymising updates preserve *inReachableState* (⇒ Sect. 3.3.5), i.e., the formula

$$inReachableState \texttt{ -> } \{\mathcal{V}(Mod)\}\ inReachableState$$

is logically valid for any $\mathcal{V}(Mod)$.

Improved Invariant Rule for JAVA CARD DL

We now present an invariant rule that makes use of the information contained in a correct modifier set (if available). This rule is an improvement over rule invRuleAnonymisingUpdate since it keeps as much of the context as possible, i.e., only locations described by the modifier set are assigned unknown values [Beckert et al., 2005b]. In contrast, rule invRuleAnonymisingUpdate assigns unknown values to *all* locations, no matter whether they can be modified by the loop or not.

The rule loopInvariantRule is identical to rule invRuleAnonymisingUpdate except that instead of the anonymising update $\mathcal{V}$, the update $\mathcal{V}(Mod)$ is used that is anonymising with respect to a modifier set *Mod* being correct for the set

```
{ ;
  if (nse) { p }
  if (nse) { p } if (nse) { p }
  if (nse) { p } if (nse) { p } if (nse) { p }
  ⋮
}
```

of programs. That is, the required modifier must be correct not only for the loop body p and the loop condition *nse* but also for an arbitrary number of iterations which, in classical dynamic logic, is denoted by $(\texttt{if (}nse\texttt{) \{ }p\texttt{ \}})^*$ using the iteration operator $*$.

loopInvariantRule

$$\frac{\begin{array}{l}\Longrightarrow Inv\\ \mathcal{U}'Inv \Longrightarrow \mathcal{U}'[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{TRUE} \texttt{ -> } ([p]Inv \,\&\, [p]_{continue}Inv))\\ \mathcal{U}'Inv,\ \mathcal{U}'\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle_{AT'}\,\text{true} \Longrightarrow \mathcal{U}'[\pi\ v\texttt{=}nse\texttt{; }\omega]\phi\\ \mathcal{U}'Inv,\ \mathcal{U}'\langle\texttt{boolean } v\texttt{=}nse\texttt{;}\rangle(v \doteq \text{TRUE} \,\&\, \langle p\rangle_{AT}\,\text{true}) \Longrightarrow \mathcal{U}'[\pi\ v\texttt{=}nse\texttt{;}p\ \omega]\phi\\ \mathcal{U}'Inv \Longrightarrow \mathcal{U}'[\texttt{boolean } v\texttt{=}nse\texttt{;}](v \doteq \text{FALSE} \texttt{ -> } [\pi\ \omega]\phi)\end{array}}{\Longrightarrow [\pi \texttt{ while (}nse\texttt{) \{ }p\texttt{ \} }\omega]\phi}$$

where $\mathcal{U}' = \mathcal{V}(Mod)$ and *Mod* is a correct modifier set for $(\texttt{if (}nse\texttt{) \{ }p\texttt{ \}})^*$.

Example 3.65. The following example shows that a "normal" modifier set that is correct for the loop condition and loop body is not sufficient for the rule to be sound.

Consider the program

JAVA

```
while ( i<10 ) {
  if ( i>0 ) {
    a = 5;
  }
  i=i+1;
}
```

JAVA

which we abbreviate with p in the following. A correct modifier set for the loop body and loop condition would be $Mod = \{\langle \text{true}, \texttt{i}\rangle, \langle \texttt{i} > 0, \texttt{a}\rangle\}$ since `i` is modified in any case and `a` is modified if `i` is greater than zero. The anonymising update with respect to *Mod* is in this case

$$\mathcal{V}(Mod) = \texttt{for}\ y_1;\ \text{true};\ \texttt{i} := c \,||\, \texttt{for}\ y_2;\ \texttt{i} > 0;\ \texttt{a} := d\ .$$

Setting $\mathcal{U}' = \mathcal{V}(Mod)$ we try to prove the (invalid) formula

$$\{\texttt{i} := 0 \,||\, \texttt{a} := 0\}\ [\pi\ p\ \omega]a \doteq 0\ .$$

Applying the loopInvariantRule with $Inv = (\texttt{a} \doteq 0)$ yields as a fifth premiss (after some simplification)

$$\{\texttt{a} := 0 \,||\, \texttt{i} := c\}\ \texttt{a} \doteq 0 \Longrightarrow \{\texttt{a} := 0 \,||\, \texttt{i} := c\}\ [\texttt{v=i<10;}](\texttt{v} \doteq \text{FALSE} \texttt{ -> } [\pi\ \omega]\texttt{a} \doteq 0)$$

which is a valid sequent. We do not show the other four premisses here which are also valid, i.e., the rule is not sound with $\mathcal{U}' = \mathcal{V}(Mod)$.

The reason for the unsoundness here is that *Mod* is a correct modifier set for the loop body and loop condition if executed only once. However, in a loop the body can be executed several times. In our example the modifier set $Mod = \{\langle \text{true}, \texttt{i}\rangle, \langle \texttt{i} > 0, \texttt{a}\rangle\}$ is correct for the program

```
if (i>0) { a=5; } i=i+1;
```

but not for

```
if (i>0) { a=5; } i=i+1; if (i>0) { a=5; } i=i+1; .
```

That is, the anonymising update $\mathcal{V}(Mod)$ only anonymises the locations that are modified in the first iteration of the loop. For the rule to be sound we however need an anonymising update that affects all locations that are changed by any execution of the body.

Usually, a modifier set describes the changes that a given, fixed (part of a) program can perform. In contrast, the modifier set that is required for the rule loopInvariantRule must describe the changes of an aribtrary number of iterations of a given program. This has two serious drawbacks: Firstly, it is unintuitive for the user since he is used to give modifier sets for *one* given program and, secondly, the proof obligation for the correctness of modifier sets (⇒ Sect. 8.3.3) cannot be used offhand for an unknown number if iterations of a program. In the following section we therefore present a method how a (correct) modifier set for the iteration p^* of a program p can be generated automatically from a (correct) modifier set for p.

Generating Modifier Sets for Iterated Programs

The following theorem states how a correct modifier set Mod_p^* for p^* can be obtained if a correct modifier set for p is given.

Theorem 3.66. *Let*

$$Mod_p = \{\langle \phi_1, f_1(s^1_1, \ldots, s^1_{n_1})\rangle, \ldots, \langle \phi_m, f_m(s^m_1, \ldots, s^m_{n_m})\rangle\}$$

be a correct modifier for the program p such that the ϕ_i ($1 \leq i \leq m$) are first-order formulae. Then the modifier set Mod^*_p that is the least set satisfying the conditions

- $Mod_p \subseteq Mod^*_p$.
- *If $\langle \psi, g(t_1, \ldots, t_n)\rangle \in Mod_p$, then $\langle \psi', g(t'_1, \ldots, t'_n)\rangle \in Mod^*_p$ if there is a substitution $\sigma = [x_1/l_1(r^1_1, \ldots, r^1_{o_1}), \ldots, x_k/l_k(r^k_1, \ldots, r^k_{o_k})]$ such that*
  - *the variables x_i are fresh and of the same type as $l_i(r^i_1, \ldots, r^i_{o_i})$,*
  - *for each $l_i(r^i_1, \ldots, r^i_{o_i})$ there is some $\langle \phi, f(s_1, \ldots, s_k)\rangle \in Mod_p$ such that $f = l$ and $k = o_i$, and*
  - *$\sigma(\psi') = \psi$ and $\sigma(g(t'_1, \ldots, t'_n)) = g(t_1, \ldots, t_n)$.*

is correct for the iteration p^* of p.

Note that the modifier set Mod^*_p is not necessary minimal, even if Mod_p is minimal for p.

In the KeY system we make use of the above theorem, i.e., when applying the rule loopInvariantRule the user is asked to provide a modifier set that is correct for the loop body p and loop condition *nse*. The system then automatically generates a modifier set for the iterated loop body.

Example 3.67. We revisit Example 3.65 and apply Theorem 3.66 in order to obtain a correct modifier set for the iterated loop body.

For the loop in Example 3.65 a correct modifier set for the loop body and the loop condition is

$$Mod = \{\langle \text{true}, \mathtt{i}\rangle, \langle \mathtt{i} > 0, \mathtt{a}\rangle\} \ .$$

Then, following Theorem 3.66, a correct modifier set for the iterated loop body is

$$Mod^* = \{\langle \text{true}, \mathtt{i}\rangle, \langle \mathtt{i} > 0, \mathtt{a}\rangle, \langle x > 0, \mathtt{a}\rangle\}$$

with the corresponding substitution $\sigma = [x/\mathtt{i}]$.

For another example consider the program

JAVA
```
while ( i<10 ) {
  ar[i]=0;
  i=i+1;
}
```
JAVA

A correct modifier set for the loop body and loop condition is

$$Mod = \{\langle \text{true}, \texttt{i}\rangle, \langle \text{true}, \texttt{ar[i]}\rangle\}$$

and, following Theorem 3.66, a correct modifier set for the iterated loop body is

$$Mod = \{\langle \text{true}, \texttt{i}\rangle, \langle \text{true}, \texttt{ar[i]}\rangle, \langle \text{true}, \texttt{ar[x]}\rangle\}$$

with the corresponding substitution $\sigma = [x/\texttt{i}]$.

3.8 Calculus Component 4: Using Method Contracts

There are basically two possibilities to deal with method calls in program verification: inlining the body of the invoked method (⇒ Sect. 3.6.5) or using the specification (which then, of course, has to be verified). The latter approach is discussed in this section.

Exploiting the specifications is indispensable in order for program verification to scale up. This way, each method only needs to be verified (i.e., executed symbolically) once. In contrast, inlined methods may have to be symbolically executed multiple times, and the size of the proofs would grow more than linearly in the size of the program code to be executed symbolically. Moreover, the source code of a (library) method may not be available. Then, the only way to deal with the invocation of the method is to use its specification.

The specification of a method is called *method contract* and is defined as follows.

Definition 3.68 (Method contract). *A method contract for a method or constructor op declared in a class or interface $C \in P$ is a quadruple*

$$(Pre, Post, Mod, term)$$

where:

- *Pre ∈ Formulae is the precondition that may contain the following program variables:*
  - *self for the receiver object (the object which a caller invokes the method on); if op refers to a static method or a constructor the receiver object variable is not allowed;*
  - *$p_1 \ldots, p_n$ for the parameters.*
- *Post ∈ Formulae is the postcondition of the form*

  $$(exc \doteq \texttt{null} \rightarrow \phi) \;\&\; (exc\,! \doteq \texttt{null} \rightarrow \psi)$$

  where ϕ is the postcondition for the case that the method terminates normally and ψ specifies the case where the method terminates abruptly with an exception. The formulae ϕ and ψ may contain the following program variables:

  - *self for the receiver object; again the receiver object variable is not allowed for static methods;*
  - $p_1, \ldots, p_n$ *for the parameters;*
  - *result for the returned value;*
- *Mod is a modifier set for the method op.*
- *The termination marker term is an element from the set* $\{partial, total\}$*; the marker is set to total if and only if the method contract requires the method or constructor to terminate, otherwise term is set to partial.*

The formulae *Pre* and *Post* are JAVA CARD DL formulae. However, in most cases they do not contain modal operators. This is in particular true if they are automatically generated translations of JML or OCL specifications.

In this section, we assume that the method contract to be considered is correct, i.e., the method satisfies its contract. This is a prerequisite for the method contract rule to be correct. The question how to establish correctness of a method contract is addressed in Sect. 8.2.4.

The rule for using a contract for a method invocation in a diamond modality looks as follows:

methodContractTotal

$$\frac{\begin{array}{l}\Longrightarrow \{self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\}\ Pre \\ \{\mathcal{V}(Mod)\}\ exc \doteq \texttt{null} \Longrightarrow \\ \quad \{\mathcal{V}(Mod) \,||\, self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n \,||\, lhs := result\} \\ \qquad (Post \texttt{ -> } \langle \pi\ \omega \rangle \phi) \\ \{\mathcal{V}(Mod)\}\ exc \mathrel{!}\doteq \texttt{null} \Longrightarrow \\ \quad \{\mathcal{V}(Mod) \,||\, self := se_{target} \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\} \\ \qquad (Post \texttt{ -> } \langle \pi\ \texttt{throw}\ exc\texttt{;}\ \omega \rangle \phi)\end{array}}{\Longrightarrow \langle \pi\ \ lhs\texttt{=}se_{target}\texttt{.}\,mname\texttt{(}se_1, \ldots, se_n\texttt{)@}C\texttt{;}\ \omega \rangle \phi}$$

where $\mathcal{V}(Mod)$ is an anonymising update w.r.t. the modifier set *Mod* of the method contract.

The above rule is applicable to a method-body-statement

$$lhs\texttt{=}se\texttt{.}\,mname\texttt{(}t_1, \ldots, t_n\texttt{)@}C\texttt{;}$$

if a contract $(Pre, Post, Mod, total)$ for the method $mname(T_1, \ldots, T_n)$ declared in class C is given. Note, that the rule cannot be applied if a contract with the termination marker set to *partial* is given since then termination of the method to be invoked is not guaranteed.

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 p_i with the receiver object *se* and with the parameters se_i. This guarantees that the method contract's precondition is fulfilled and we can use the postcondition *Post* to describe the effect 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 program variable *exc* is `null`. If the method

is non-void the return value *return* is assigned to the variable *lhs*. The second case deals with the situation that the method terminates abruptly (third premiss). Note, that in both cases the postcondition *Post* holds and the locations that the method possibly modifies are updated with the anonymising update $\mathcal{V}(Mod)$ with respect to *Mod*. As in the first premiss, the variables for the receiver object and the parameters are updated with the corresponding terms. In case of abrupt termination there is no result but an exception *exc* that must be thrown explicitly in the fourth premiss to make sure that the control flow of the program is correctly reflected.

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

methodContractPartial

$$\frac{\begin{array}{l}\Longrightarrow \{self := se \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n\}\ Pre \\ exc \doteq \texttt{null} \Longrightarrow \\ \quad \{\mathcal{V}(Mod) \,||\, self := se \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := se_n \,||\, lhs := result\} \\ \qquad (Post \texttt{ -> } [\pi\ \omega]\phi) \\ exc \mathrel{!}\doteq \texttt{null} \Longrightarrow \\ \quad \{\mathcal{V}(Mod) \,||\, self := se \,||\, p_1 := se_1 \,||\, \cdots \,||\, p_n := t_n\} \\ \qquad (Post \texttt{ -> } [\pi\ \texttt{throw } exc\texttt{; } \omega]\phi)\end{array}}{\Longrightarrow [\pi\ \ lhs\texttt{=}se\texttt{.}mname\texttt{(}se_1, \ldots, se_n\texttt{)@}C\texttt{; } \omega]\phi}$$

where $\mathcal{V}(Mod)$ is an anonymising update w.r.t. the modifier set *Mod* of the method contract.

Example 3.69. The following JAVA CARD class `MCDemo` contains the two methods `inc` and `init` that are annotated with JML specifications. The contract of `inc` states that:

- The method terminates normally.
- The result is equal to the sum of the parameter `x` and the literal 1.
- The method is pure, i.e., does not modify any location.

The contract of `init` expresses that:

- The method terminates normally.
- When the method terminates, the result is equal to the sum of the parameter `u` and the literal 1 and the attribute `attr` has the value 100

JAVA

```
public class MCDemo {

    int attr;

  /*@ public normal_behavior
    @ assignable \nothing;
    @ ensures \result == x+1;
```

```
    @ */
  public int inc(int x) {
      return ++x;
  }



  /*@ public normal_behavior
    @    ensures \result == u+1 && attr == 100;
    @ */
  public int init(int u) {
      attr = 100;
      return inc(u);
  }

}
```

JAVA

In this example, we want to prove the (total) correctness of the method `init`. We feed this annotated JAVA CARD class into the JML front-end of the KeY system and select the corresponding proof obligation for total correctness. When we arrive at the point in the proof where method `inc` is invoked we apply the rule methodContractTotal. This is possible even if we have not explicitly set-up a method contract according to Def. 3.68. Rather the KeY system automatically translates the JML specification of method `inc` into the method contract $(Pre, Post, Mod, term)$ with

$$\begin{aligned} Pre &= \text{true} \\ Post &= result \doteq p_1 + 1 \\ Mod &= \{\} \\ term &= total \end{aligned}$$

Since we have specified that the method `inc` terminates normally, the KeY system generates only the following (slightly simplified) two premisses instead of the three in the rule scheme (the one dealing with abrupt termination is omitted):

1. In the first sequent we have to establish the precondition of method `inc` in the state where `inc` is invoked. This is trivial here since the JML specification does not contain a `requires` clause and, thus, the precondition is true.

   KeY

   ```
   ==>
    {u_old:=u_lv,
      self:=self_lv,
   ```

```
x:=u_lv,
self_lv.attr:=100} true
```

KeY

2. In the second sequent we make use of the postcondition of method `inc`. In lines 4 and 5 the variables for the receiver object `self` and the parameter `x`, respectively, are updated with the corresponding parameter terms. In the JML specification we have an `assignable nothing;` statement saying that the method `inc` does not modify any location. As a consequence, the anonymising update $\mathcal{V}(Mod)$ in the rule scheme here is empty and, thus, omitted.

KeY

```
==>
 {u_old:=u_lv,
3  self_lv.attr:=100
  self:=self_lv,
  x:=u_lv}
6   (   j = x + 1
     -> \<{method-frame(result->result,
                        source=MCDemo,
9                       this=self): {
              return  j;
            }
12        }\> (result = u_old + 1  &  self.attr = 100))
```

KeY

The validity of the two sequents shown above can be established automatically by the KeY prover.

Note that the program would also be correct if we omit from the specification `assignable nothing;`. Then, however, we could not prove the correctness of the program using the rule methodContractTotal since for the rule being sound it must be assumed that anything, i.e., in particular the attribute `attr`, can be changed by the corresponding method. As a consequence, the validity of the equation `attr = 100` in the formula following the diamond modality in the above sequent cannot be established. If instead of the method contract rule the rule for inlining the method body is used, the correctness of the program can be shown even if the assignable clause is missing since the implementation of method `inc` in fact does not modify the attribute `attr`.

3.9 Calculus Component 5: Update Simplification

The process of update simplification comprises (a) update normalisation and (b) update application. Update normalisation transforms single updates into

a certain normal form, while update application involves an update and a term, a formula, or another update that it is applied to. Note that in the KeY system both normalisation and application of updates is done automatically; there are no interactive rules for that purpose.

3.9.1 General Simplification Laws

We first define an equivalence relation on the set of JAVA CARD DL updates, which holds if and only if two updates always have the same effect, i.e., represent the same state transition.

Definition 3.70. *Let u_1, u_2 be JAVA CARD DL updates. The relation*

$$\equiv \subseteq \mathit{Updates} \times \mathit{Updates}$$

is defined by

$$u_1 \equiv u_2 \quad \textit{iff} \quad \mathrm{val}_{S,\beta}(u_1) = \mathrm{val}_{S,\beta}(u_2)$$

for all variable assignments β and JAVA CARD DL states S.

The first update simplification law expressed in the following lemma is that the sequential and parallel update operators are associative.

Lemma 3.71. *For all $u_1, u_2, u_3 \in \mathit{Updates}$ the following holds:*

- $u_1 \,||\, (u_2 \,||\, u_3) \equiv (u_1 \,||\, u_2) \,||\, u_3$,
- $u_1 \,;\, (u_2 \,;\, u_3) \equiv (u_1 \,;\, u_2) \,;\, u_3$.

This justifies that in the sequel we omit parentheses when writing lists of sequential or parallel updates. However, neither of the operators $||$ and $;$ is commutative, as the following example demonstrates.

Example 3.72. The sequential and parallel update operators are not commutative:

$$\begin{array}{lcllcl} i := 0 \,;\, i := 1 & \equiv & i := 1 & \not\equiv & i := 0 & \equiv & i := 1 \,;\, i := 0 \\ i := 0 \,||\, i := 1 & \equiv & i := 1 & \not\equiv & i := 0 & \equiv & i := 1 \,||\, i := 0 \end{array}$$

Another simple law is that quantification in an update has no effect if the quantified variable does not occur in the scope.

Lemma 3.73. *Let x be a variable, ϕ a JAVA CARD DL formula, and u an update. If ϕ is logically valid and $x \notin \mathit{fv}(\phi) \cup \mathit{fv}(u)$ then*

$$(\texttt{for}\ x;\ \phi;\ u) \quad \equiv \quad u \ .$$

3.9.2 Update Normalisation

In the following we present a normal form for updates and explain how arbitrary updates are transformed into this normal form. We use "`for` $\bar{x}$; ϕ; u" and "`for` $(x_1, \ldots, x_n)$; ϕ; u" to abbreviate "`for` x_1; true; $\cdots$ `for` x_n; ϕ; u".

The normal form for updates is a sequence of quantified updates (with function updates as sub-updates) executed in parallel.

Definition 3.74 (Update Normal Form). *An update u is in* update normal form *if it has the form*

$$\texttt{for}\ \bar{x}_1;\ \phi_1;\ u_1 \,||\, \texttt{for}\ \bar{x}_2;\ \phi_2;\ u_2 \,||\, \cdots \,||\, \texttt{for}\ \bar{x}_n;\ \phi_n;\ u_n$$

where the u_i are function updates (⇒ Def. 3.8).

It is crucial for this normal form that the well-ordering of the domain (of a JAVA CARD DL Kripke structure with ordered domain) is expressible in the object logic. For that purpose JAVA CARD DL contains the binary predicate *quanUpdateLeq* . It is used for resolving clashes in quantified updates on a syntactic level. This requires to express that there is an element x satisfying some property ϕ and that it is the smallest such element: $\exists x.(\phi \;\&\; \forall y.([y/x]\phi \;->\; \mathit{quanUpdateLeq}(x, y)))$.

We now present simplification laws that allow arbitrary updates to be turned into normal form.

Function Updates

A function update $f(t_1, \ldots, t_n) := s$ can easily be transformed into normal form by applying Lemma 3.73:

$$f(t_1, \ldots, t_n) := s \quad \equiv \quad \texttt{for}\ x;\ \text{true};\ f(t_1, \ldots, t_n) := s$$

where $x \notin \mathit{fv}(f(t_1, \ldots, t_n) := s)$.

Sequential Update

Sequential updates $u_1 \,; u_2$ can be transformed into normal form by applying the following law which introduces an update application $\{u_1\}\ u_2$ (⇒ Sect. 3.9.3).

Lemma 3.75. *For all $u_1, u_2 \in$ Updates:*

$$u_1 \,; u_2 \quad \equiv \quad u_1 \,||\, \{u_1\} u_2$$

Quantified Updates with Non-function Sub-updates

We consider a quantified update `for` x; ϕ; u where u is not a function update (otherwise the update would already be in normal form).

If u is a sequential update, we apply the previous rule to transform u into a parallel update. The handling of parallel updates however is not that straightforward. For $u = u_1 \parallel u_2$, the quantification cannot be simply distributed over the parallel update operator as the following example shows.

Example 3.76. For simplicity, we assume that x ranges only over the non-negative integers (which shall be ordered as usual). Then,

$$
\begin{aligned}
&\texttt{for } x;\ 0 <= x <= 2;\ (f(x+1) := x \parallel f(x) := x) \\
&\quad \equiv \quad f(3) := 2 \parallel f(2) := 2 \parallel f(2) := 1 \parallel f(1) := 1 \parallel f(1) := 0 \parallel f(0) := 0 \\
&\quad \equiv \quad f(3) := 2 \parallel f(2) := 1 \parallel f(1) := 0 \parallel f(0) := 0 \\
&\quad \not\equiv \quad f(3) := 2 \parallel f(2) := 2 \parallel f(1) := 1 \parallel f(0) := 0 \\
&\quad \equiv \quad f(3) := 2 \parallel f(2) := 1 \parallel f(1) := 0 \parallel f(2) := 2 \parallel f(1) := 1 \parallel f(0) := 0 \\
&\quad \equiv \quad (\texttt{for } x;\ 0 <= x <= 2;\ f(x+1) := x) \parallel \\
&\qquad\quad (\texttt{for } x;\ 0 <= x <= 2;\ f(x) := x)
\end{aligned}
$$

As the above example suggests, a quantified update `for` x; ϕ; u can be understood as a (possibly infinite) sequence $\ldots \parallel [x/t_2]u \parallel [x/t_1]u$ where instances of the sub-update u are put in parallel for all values satisfying the guard (syntactically represented by terms t_i). To preserve the clash semantics of quantified updates, the order of the updates $[x/t_i]u$ put in parallel is crucial. A term t_i must evaluate to a domain element d_i that is smaller than or equal to all the d_j that the terms t_j, $j > i$, evaluate to. Intuitively, in the sequence $\ldots \parallel [x/t_2]u \parallel [x/t_1]u$ a term t_i must be smaller than all the terms t_{i+n} occurring to its left to correctly represent the corresponding quantified update, since this guarantees that in case of a clash "the least element wins".

Distributing a quantification over the parallel-composition operator corresponds to a permutation of the updates in the sequence $\ldots \parallel [x/t_2]u \parallel [x/t_1]u$, which in general alters the semantics (as the above example shows). Only in the case that no clashes occur, permutations preserve the semantics of parallel updates.

Example 3.77. We revisit the updates from Example 3.76 and visualise the permutation of sub-updates induced by distributing quantification over the parallel-composition operator. The arrows in Fig. 3.11 indicate the order of the updates $[x/t_i]f(x+1) := x$ and $[x/t_i]f(x) := x$ if the quantified updates from Example 3.76 are understood as a sequence of parallel updates.

The left part of Fig. 3.11 shows the order for the update

$$\texttt{for } x;\ 0 \leq x \leq 2;\ (f(x+1) := x \parallel f(x) := x)$$

and the right part the order for

$$\texttt{for } x;\ 0 \leq x \leq 2;\ f(x+1) := x \,||\, \texttt{for } x;\ 0 \leq x \leq 2;\ f(x) := x \ ,$$

i.e., after distributing the quantification over the parallel sub-update. The figure shows that the order of clashing parallel updates in the two cases differ. For example, the update $[x/1](f(x) := x)$ clashes with $[x/0](f(x+1) := x)$. In the left part of the figure, the latter update wins, while in the right part, the former update takes precedence.

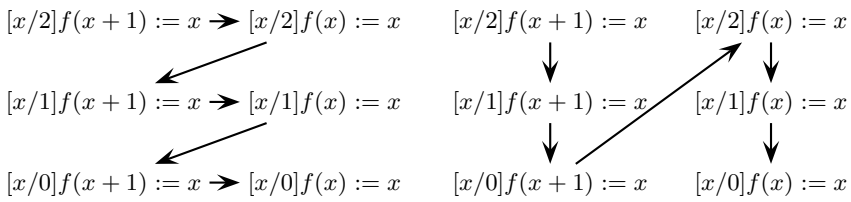


Fig. 3.11. Evaluation order of quantified updates

Fig. 3.11 does not only illustrate that naive distribution of quantification over parallel composition is not a correct update simplification law, but also gives a hint on how to "repair" the law based on the following two observations:

- If no clashes occur, the parallel update operator is commutative, i.e., in that case the order in a sequence of parallel updates is irrelevant.
- In case of a clash, the update that gets overridden by a later update can simply be omitted (since parallel updates do not influence each other).

The idea is now to distribute the quantification and to add a guard formula ψ to the quantified update to prevent wrong overriding. The formula ψ is constructed in such a way that it evaluates to false if the update would wrongly override another update. For example, in Fig. 3.11 that situation occurs if an update clashes with an other update that is located in a column to its left and in a row below.

That way, an update of the form

$$\texttt{for } x;\ \varphi;\ (u_1 \,||\, u_2)$$

can still be transformed into an update of the shape

$$\texttt{for } x;\ \varphi;\ u_1 \,||\, \texttt{for } x;\ \varphi \,\&\, \psi;\ u_2$$

where u_1 can be assumed to be in normal form

$$\texttt{for } \bar{y}_1;\ \varphi_1;\ v_1 \,||\, \texttt{for } \bar{y}_2;\ \varphi_2;\ v_2 \,||\, \cdots \,||\, \texttt{for } \bar{y}_n;\ \varphi_n;\ v_n$$

and u_2 is an update of the form

$$\texttt{for}\ (z_1, \ldots, z_o);\ \omega;\ f(t_1, \ldots, t_k) := s\ .$$

The formula ψ which adds additional constraints preventing the right update from overriding the left one in a wrong way looks like

$$\psi = \forall x'.((C_1 \mid \cdots \mid C_n) \texttt{ -> } \mathit{quanUpdateLeq}(x, x'))$$

where x' is a fresh variable and C_i determines whether the i-th part

$$\texttt{for}\ (y_1, \ldots, y_l);\ \varphi_i;\ g(r_1, \ldots, r_m) := s_i$$

of u_1 might collide with u_2.

This is the case if and only if

- the left hand sides $f(t_1, \ldots, t_k)$ and $g(r_1, \ldots, r_m)$ of both updates syntactically match (i.e. same top-level function symbols $f = g$ and same arities $k = m$) and
- there are values $y_1, \ldots, y_l$ such that the guard φ_i evaluates to true for a value $x' < x$ (i.e. u_2 illicitly overrides u_1 "in a row below") and for that same value x' the arguments t_j and r_j pairwise evaluate to the same value (i.e. there is in fact a clash).

Thus, C_i is defined as

$$\begin{aligned} C_i &= \begin{cases} \exists y_1. \cdots \exists y_l. C_i' & \text{for } f = g, k = m \\ \text{false} & \text{otherwise} \end{cases} \\ C_i' &= [x/x']\varphi_i \ \&\ t_1 \doteq [x/x']r_1 \ \&\ \cdots \ \&\ t_k \doteq [x/x']r_k \end{aligned}$$

Example 3.78. We apply the transformation described above to Example 3.76. Since we assume x to range only over non-negative integers with the usual ordering, we can write $y <= z$ instead of $\mathit{quanUpdateLeq}\ (y, z)$.

In a first step we transform the sub-updates of the parallel update $f(x+1) := x \,||\, f(x) := x$ into normal form:

$$\begin{aligned} &\texttt{for}\ x;\ 0 <= x <= 2;\ (f(x+1) := x \,||\, f(x) := x) \\ &\quad \equiv \quad \texttt{for}\ x;\ 0 <= x <= 2;\ (\texttt{for}\ y;\ \text{true};\ f(x+1) := x \,|| \\ &\qquad\qquad\qquad\qquad\qquad\qquad \texttt{for}\ z;\ \text{true};\ f(x) := x) \end{aligned}$$

Then, we distribute the quantification over the parallel update and add a formula ψ to guarantee the correctness of the transformation.

$$\begin{aligned} \equiv \quad & (\texttt{for}\ x;\ 0 <= x <= 2;\ \texttt{for}\ y;\ \text{true};\ f(x+1) := x) \,|| \\ & (\texttt{for}\ x;\ 0 <= x <= 2\ \&\ \psi;\ \texttt{for}\ z;\ \text{true};\ f(x) := x) \end{aligned}$$

where $\psi = \forall x'.(x \doteq x' + 1 \texttt{ -> } x <= x')$. The formula $0 <= x <= 2\ \&\ \psi$ can be simplified to $x \doteq 0$, and we obtain

$$
\begin{aligned}
&\equiv \quad (\texttt{for}\ x;\ 0 <= x <= 2;\ \texttt{for}\ y;\ \text{true};\ f(x+1) := x)\ ||\\
&\qquad (\texttt{for}\ x;\ x \doteq 0;\ \texttt{for}\ z;\ \text{true};\ f(x) := x)\\
&\equiv \quad (\texttt{for}\ x;\ 0 <= x <= 2;\ f(x+1) := x)\ ||\\
&\qquad (\texttt{for}\ x;\ x \doteq 0;\ f(x) := x)\\
&\equiv \quad f(3) := 2\ ||\ f(2) := 1\ ||\ f(1) := 0\ ||\ f(0) := 0
\end{aligned}
$$

The last equivalence shows that the transformed formula is in fact equivalent to the original one (see Example 3.76).

Note 3.79. The normal form for updates consists of quantified updates put in parallel. In KeY we also allow function updates to appear instead of quantified updates, i.e., it is not necessary to transform a function update into a quantified update. The reason is that the majority of updates are function updates and the normal form becomes very clumsy and hard to read if these updates are transformed into quantified updates (see, e.g., Example 3.76).

In the KeY system, the parallel sub-updates of an update in normal form are ordered lexicographically. That makes it possible to close many proof goals without additional rules for permuting the parallel sub-updates.

3.9.3 Update Application

The second part of the update simplification process is the application of updates to other updates, terms, and formulae. Since updates are "semantical" substitutions, the application of an update cannot (always) be effected by a mere syntactical substitution but may require more complex syntactical manipulations.

Applying an Update to an Update

The application of an update to another update is based on the following simplification laws.

Lemma 3.80. *Let an arbitrary update $u \in$ Updates and function updates $u_1, \ldots, u_n \in$ Updates be given. Then,*

- $\{u\}\ (f(t_1, \ldots, t_n) := s) \quad \equiv \quad f(\{u\}\ t_1, \ldots, \{u\}\ t_n) := \{u\}\ s$
- *if none of the variables in the variable lists $\bar{x}_i$ occur in u*

$$
\begin{aligned}
&\{u\}\ (\texttt{for}\ \bar{x}_1;\ \phi_1;\ u_1\ ||\ \cdots\ ||\ \texttt{for}\ \bar{x}_n;\ \phi_n;\ u_n) \quad \equiv\\
&\texttt{for}\ \bar{x}_1;\ \{u\}\ \phi_1;\ \{u\}\ u_1\ ||\ \cdots\ ||\ \texttt{for}\ \bar{x}_n;\ \{u\}\ \phi_n;\ \{u\}\ u_n
\end{aligned}
$$

In the above lemma only applications of updates to function updates and to updates in normal form are considered. That, however, is sufficient since all updates can be transformed into normal form using the rules from Sect. 3.9.2.

Applying an Update to a Term

In the following, we use the notation $t \equiv t'$ and $\phi \equiv \phi'$ to denote that the terms t, t' resp. the formulae ϕ, ϕ' have the same value in all states for all variable assignments, in which case one can safely be replaced by the other preserving the semantics of the term of formula.

Definition 3.81. *Given terms $t, t' \in$ Terms and formulae $\phi, \phi' \in$ Formulae, we write*

- $t \equiv t'$ *if the formula $t \doteq t'$ is logically valid,*
- $\phi \equiv \phi'$ *if the formula $\phi \leftrightarrow \phi'$ is logically valid.*

Lemma 3.82. *Let*

$$u = \texttt{for } \bar{y}_1;\ \phi_1;\ t_1 := s_1 \parallel \cdots \parallel \texttt{for } \bar{y}_m;\ \phi_m;\ t_m := s_m$$

be an update in normal form. Then,

- *for all rigid terms $t \in$ Terms,*

$$\{u\}\, t \quad \equiv \quad t \ ,$$

- *for all terms $f(a_1, \ldots, a_n) \in$ Terms,*

$$\begin{array}{l} \{u\}\, f(a_1, \ldots, a_n) \quad \equiv \\ \text{if } C_m \text{ then } T_m \text{ else } \ldots \text{ if } C_1 \text{ then } T_1 \text{ else } f(\{u\}\, a_1, \ldots, \{u\}\, a_n) \end{array}$$

where $C_1, \ldots, C_m$ are guard formulae expressing that the i-th sub-update of u affects the term $f(a_1, \ldots, a_n)$, and $T_1, \ldots, T_m$ are terms that describe the value of the expression in these cases.
C_i and T_i are defined as follows. Suppose that the i-th part of u is of the form

$$\texttt{for } (z_1, \ldots, z_l);\ \phi_i;\ g(b_1, \ldots, b_k) := s_i \ .$$

Then, the formula C_i is defined by

$$\begin{array}{rcl} C_i & = & \begin{cases} \exists z_1. \cdots \exists z_l.\ C_i' & \text{if } f = g \text{ and } n = k \\ \text{false} & \text{otherwise} \end{cases} \\ C_i' & = & \phi_i \ \&\ (\{u\}\, a_1) \doteq b_1 \ \&\ \cdots \ \&\ (\{u\}\, a_k) \doteq b_k \end{array}$$

and the terms T_i are constructed from the s_i by applying substitutions that instantiate the occurring variables with the smallest of clashing values (corresponding to the clash semantics of quantified updates):

$$\begin{array}{l} \big[z_1 \,/\, (\text{ifExMin } z_1.\exists z_2. \cdots \exists z_l.\ C_i' \text{ then } z_1 \text{ else } z_1), \\ \ z_2 \,/\, (\text{ifExMin } z_2.\exists z_3. \cdots \exists z_l.\ C_i' \text{ then } z_2 \text{ else } z_2), \\ \ \ldots, \\ \ z_l \,/\, (\text{ifExMin } z_l.C_i' \text{ then } z_l \text{ else } z_l)\big]\, s_i \end{array}$$

- *for all* $u_1 \in \mathit{Updates}$ *and* $t \in \mathit{Terms}$,

$$\{u\}\,(\{u_1\}\,t) \quad \equiv \quad \{u\,;u_1\}\,t \ .$$

The order of the C_i and T_i in the second equivalence in the above lemma is relevant. Due to the last-win semantics of parallel updates, the right-most sub-update $\texttt{for}\ \bar{y}_i;\ \phi_i;\ t_i := s_i,\ i = m$, must be checked first, and the left-most sub-update, $i = 1$, must be checked last such that the i-th update "wins" over the j-th update if $i > j$.

Example 3.83. As an example, we consider the term $\{a(o) := t\}\ a(p)$. Intuitively, the update $a(o) := t$ affects the term $a(p)$ iff o and p evaluate to the same domain element. In a first step, we transform the update into normal form:

$$a(o) := t \quad \equiv \quad \texttt{for}\ y;\ \mathrm{true};\ a(o) := t$$

where y is a fresh variable. Now, we can apply the normalised update on the term $a(p)$ using Lemma 3.82:

$$\begin{array}{l}\{\texttt{for}\ y;\ \mathrm{true};\ a(o) := t\}\ a(p) \quad \equiv \\ \mathrm{if}\ \ C\ \ \mathrm{then}\ \ T\ \ \mathrm{else}\ \ a(\{\texttt{for}\ y;\ \mathrm{true};\ a(o) := t\}\ p)\end{array}$$

where

$$\begin{array}{rcll} C & = & \exists y.C' & \\ C' & = & \mathrm{true}\ \&\ (\{\texttt{for}\ y;\ \mathrm{true};\ a(o) := t\}\ p) \doteq o & \\ & \equiv & (\{\texttt{for}\ y;\ \mathrm{true};\ a(o) := t\}\ p) \doteq o & \\ T & = & [y\,/\,(\mathrm{ifExMin}\,z.C'\ \ \mathrm{then}\ \ z\ \ \mathrm{else}\ \ y)]t & \\ & = & t & (\text{since } y \text{ does not occur in } t) \end{array}$$

The simplification of $(\{\texttt{for}\ y;\ \mathrm{true};\ a(o) := t\}\ p)$ yields p since it can be excluded syntactically that this update can affect the non-rigid constant p. Thus, we finally obtain

$$\begin{array}{l}\{\texttt{for}\ y;\ \mathrm{true};\ a(o) := t\}\ a(p) \quad \equiv \\ \mathrm{if}\ \ p \doteq o\ \ \mathrm{then}\ \ t\ \ \mathrm{else}\ \ a(p)\end{array}$$

which coincides with our intuition.

Applying an Update to a Formula

The following lemma contains simplification laws for applications of updates to formulae. Updates can be distributed over logical operators (except modal operators) as (a) the semantics of logical operators is not affected by a state change (b) the state change affected by an update is deterministic.

Lemma 3.84. *Let $u \in$ Updates be an update:*

- $\{u\}\ p(t_1, \ldots, t_n) \equiv p(\{u\}\ t_1, \ldots, \{u\}\ t_n)$,
- $\{u\}\ \text{true} \equiv \text{true}$ *and* $\{u\}\ \text{false} \equiv \text{false}$,
- $\{u\}\ (!\,\phi) \equiv\ !\{u\}\ \phi$,
- $\{u\}\ (\phi \circ \psi) \equiv \{u\}\ \phi \circ \{u\}\ \psi$ *for* $\circ \in \{|, \&, ->\}$,
- $\{u\}\ \forall x.\phi \equiv \forall x.\{u\}\ \phi$ *and* $\{u\}\ \exists x.\phi \equiv \exists x.\{x\}\ \phi$ *provided that* $x \notin fv(u)$,
- $\{u\}\ (\{u_1\}\ \phi) \equiv \{u\,;u_1\}\ \phi$.

The application of an update u to a formula with a modal operator, such as $\{u\}\ \langle p\rangle\phi$ and $\{u\}\ [p]\phi$, cannot be simplified any further. In such a situation, instead of using update simplification, the program p must be handled first by symbolic execution. Only when the whole program has disappeared, the resulting updates can be applied to the formula ϕ.

3.10 Related Work

An object-oriented dynamic logic, called ODL, has been defined [Beckert and Platzer, 2006], which captures the essence of JAVA CARD DL, 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 non-rigid functions that can be used to represent object attributes. However, it does not include the many other language features, built-in operators, etc. of JAVA. Using such a minimal extension that is not cluttered with too many features makes theoretical investigations much easier. A case in point are paper-and-pencil soundness and completeness proofs for the ODL calculus, which are—though not trivial—still readable, understandable and, hence, accessible to investigation.

A version of dynamic logic is also used in the software verification systems KIV [Balser et al., 2000] and VSE [Stephan et al., 2005] for (artificial) imperative programming languages. More recently, the KIV system also supports a fragment of the JAVA language [Stenzel, 2005]. In both systems, DL was successfully applied to verify software systems of considerable size.

The LOOP tool [Jacobs and Poll, 2001, van den Berg and Jacobs, 2001] translates JAVA programs and specifications written in the Java Modeling Language (JML) into proof goals expressed in higher-order logic. LOOP serves as a front-end to a theorem prover (PVS or Isabelle), in which the actual verification of the program properties takes place, based on a semantics of sequential JAVA that is formalised using coalgebras.

The JIVE tool [Meyer and Poetzsch-Heffter, 2000] follows a similar approach, translating programs that are written in a core subset of JAVA together with their specification into higher-order proof goals. These proof goals can then be discharged using the interactive theorem prover Isabelle.