Handling of Integer Arithmetic in the Verification of Java Programs

Diploma Thesis

Extended Version

University of Karlsruhe
Department of Computer Science
Institute for Logic, Complexity and Deduction Systems

Responsible Supervisor: Prof. Dr. P. H. Schmitt
Supervisor: Dr. B. Beckert

Steffen Schlager

31.5.2002

Contents

List of Figures

Preface

Topic

This diploma thesis deals with the handling of integer arithmetic in the verification of Java programs. In particular, problems like overflow and underflow arising from the finiteness of the Java types should be tackled. The aim is not only giving a calculus for Java integer arithmetic but also implementing and integrating the calculus into the theorem prover of the KeY system.

Structure of the Thesis

After a short introduction in Chapter 1, in Chapter 2 important notions and the used Dynamic Logic are defined. In Chapter 3, we dwell on the connection of specification, implementation, and verification before in Chapter 4, we define an extension of the syntax and semantics of Java integer arithmetic which the calculus, given in chapter 6, is based on. The motivation for the extension of Java syntax and semantics is presented in Chapter 5. Finally, in Chapter 7, we present the implementation and integration of the rules of the calculus into the KeY system before in Chapter 8, we give an extended example illustrating the usefulness of our approach. In the end, we draw conclusion and give some ideas for future work.

Acknowledgements

I would like to thank all the people that have supported and encouraged me in the development of this thesis. Especially, I thank my supervisors Prof. Dr. Peter H. Schmitt and Dr. Bernhard Beckert for their advice and guidance during this thesis. I thank Bernhard Beckert for the interesting topic and for many fruitful and interesting discussions. I also like to thank my fellow student assistants Richard Bubel and Andreas Roth for their indispensable support during the implementation and finally, I thank all the people working in the research group of Prof. Dr. Wolfram Menzel and Prof. Dr. Peter H. Schmitt.

1 Introduction

This thesis is carried out as part of the KeY project[1] [Ahrendt et al., 2000]. The aim of the KeY project is the integration of formal methods into the real-world software development process. The starting point of this project was the observation that formal methods are still very rarely used in practice although several case studies have shown the feasibility of computer-aided specification and verification. The reason for that is that the existing systems demand a well-founded knowledge in logic and interactive theorem proving since the used specification languages are often very mathematical and the provers are not really very user-friendly. Another obstacle for the practical use is that most existing tools are not designed for verifying programs in the programming languages that are used in practice. Rather these tools are based on artificial languages that are simple, non object-oriented, and do not offer features that make programming more convenient. Thus, since the KeY project aims at promoting the use of formal methods in practice, the main requirements for the success of the project are to chose a common and modern programming language as the target language for verification and to integrate the KeY system into a common CASE tool which helps to reduce reservations against the use of formal methods. Furthermore, the user-interface should be user-friendly and should support the user during interactive proving, for example, by offering rules that are applicable at the moment.

As the target programming language for the KeY project, Java Card was chosen. Java Card is a subset of Java which is used for programming smart cards. Applications on smart cards are often security critical which is a good incentive for program verification. Another advantage of Java Card is that it lacks some crucial complications of full Java like multi-threading, garbage collection, many library classes and data types. Furthermore, Java Card programs are usually very small due to the small memory on smart cards (current smart cards offer only 16 kilo-byte memory for code).

Another principle of the KeY project is to integrate specification, implementation, and verification into one tool. The starting point for the KeY project is the commercial CASE tool Together Control Center from TogetherSoft which offers both specification with UML/OCL and implementation in Java or C++. One reason, why we decided to chose this CASE tool is, that it is highly extensible and the verification component can by plugged in as an additional module.

[1] See also the KeY project web page under the URL `http://i12www.ira.uka.de/~key`

A design decision in the KeY project was to develop a custom-made theorem prover and not to utilise a ready to use theorem prover like Isabelle [Paulson, 1994] or PVS [Owre et al., 1992], only to mention some of the numberless available systems. Developing a theorem prover of course requires dealing with arithmetic and especially integer arithmetic, what is the topic of this thesis.

In most publications about Java and especially the semantics of Java (for example [von Oheimb, 2001, Huisman, 2001, Beckert, 2001]), the handling of integer arithmetic is missed out. For example, M. Huisman writes in her PhD-thesis:

> For the time being we abstract away from precise number representation, for example we do not deal with integer bounds, and range and precision of floating point numbers. Incorporating this requires some care, to ensure that no problems occur in theorem proving.

We think this problem should definitely be addressed in the field of verifying security relevant software, because bugs in this kind of software can result in immense economic damage or even, much worse, in danger for human lives.

2 Preliminaries

You do not really understand something unless
you can explain it to your grandmother.
— ALBERT EINSTEIN

2.1 Notational Conventions

In this work we will obey the following notational conventions:

- Numbers in binary representation are denoted by the subscript B. For example, the binary representation of 7 is 0111_B.
- JAVA programs or parts of JAVA programs are written in `type writer` font.
- We will often omit class and method declarations of JAVA programs when we only refer to a small program fragment. We will call those fragments "programs" as well although they are of course no well-formed JAVA programs.

2.2 First-order Dynamic Logic for Java

In this section we define syntax and semantics of a first-order Dynamic Logic (DL) based on the programming language JAVA with some extensions defined in Chapter 4 and some restrictions discussed in the following. We call this version of Dynamic Logic DLJ. Giving the following definitions of syntax and semantics, we follow the presentation in [Beckert, 2001] and explicitly label extensions and changes.

2.2.1 Syntax

The definition of the syntax of DLJ consists of two parts. First, we define the syntax of the programs that may appear in a DLJ formula. This is the dynamic part. Secondly, we define the non-dynamic part which is basically a typed first-order predicate logic. Finally, we have to define how DLJ formulas are built-up from programs and predicate logical formulas.

Context

The programs that may appear in a DLJ formula are basically syntactically legal single-thread JAVA programs with some modifications that are discussed later (for example they may contain our additional types defined in Chapter 4 or, under certain circumstances, logical terms). But the programs are not allowed to contain class or interface definitions. Class and interface definitions are part of the so called *context*. Since programs contained in DLJ formulas are always defined with respect to a given context, it is clear which classes and interfaces exist and how the inheritance tree is built-up.

The context is just a set of class and interface definitions with some restrictions. The definition of *local inner classes* is not allowed in the context and `break` and `continue` statements may only be used with explicit labels. But this restrictions are "harmless" because any JAVA program can be transformed in a semantically equivalent program that meets the stipulations of the context. The detailed transformation is described in [Beckert, 2001].

We assume that every class *`Cls`* defined in the context implicitly contains some additional fields and methods which may be referred by programs appearing in DL formulas but not in the context itself. These additions allow to keep track of the set of generated object and the initialising state of the object itself and its class. This is necessary because this information is not accessible in JAVA otherwise. The objects of a certain class are considered to be stored as an infinite concatenated list. In order to keep this list up to date, we assume that the semantics of the `new` operator has been extended and each application of `new` changes the attribute `lastCreatedObject` of the class and sets the attribute `created` of the new object to `true`. Figure 2.1 shows the implicitly assumed fields and methods of each Class *`Cls`* and Figure 2.2 shows the assumed semantics of the `new` operators by means of an implementation in JAVA.

2.2.2 Types of DLJ

Our version of Dynamic Logic DLJ is a typed logic containing the types defined in definition 2.2.1. The set of types used in our work extends the set of types defined in [Beckert, 2001] by the primitive JAVA types `int` and `long` that are not available in JAVA CARD[1] and by our additional primitive types defined in Chapter 4. According to the JAVA language specification, the types `int`, `short`, `int`, and `long` are called *primitive integral types*.

Definition 2.2.1 (Types of DLJ)
The set of types of DLJ *contains*

- *the primitive* JAVA CARD *types* `boolean`*,* `byte`*, and* `short`*,*

[1] The DL defined in [Beckert, 2001] is based on a subset of JAVA, called JAVA CARD.

```
public static Cls firstObj; //the first object in the list,
                            //whether already created or not

public static Cls lastCreatedObj; //the last object,
                                  // null if no object exists

public Cls prevObj; //the previous object in the list,
                    //null if for the first object

public Cls nextObj; //the next object in the list


public boolean beforeObj(Cls obj); //returns true if this
                                   //is before obj in the list

public boolean created; //returns true if the object has already been
                        //created with new, and false otherwise

public static boolean classInitialised; //returns true if the class is
                                        //initialised and false otherwise

public boolean objInitialised; //returns true if the object is
                               //initialised and false otherwise
```

Figure 2.1: This figure shows the fields and methods that are assumed to be implicitly defined for each class *`Cls`*.

- *the primitive* JAVA *types* `int` *and* `long` *(extension of [Beckert, 2001]),*
- *the additional types* `arithByte`*,* `arithShort`*,* `arithInt`*, and* `arithLong` *defined in Chapter 4 (extension of [Beckert, 2001]),*
- *the classes defined in the context,*
- *the type* Null*,*
- *the built-in classes such as* `Object` *and* `String`*,*
- *an array type* *`T[]`* *for each built-in and additional primitive type, for each array type, and for each class, and*
- *user-defined types (typically as abstract data types (ADTs)).*

```
public static Cls new(){
  if (lastCreatedObj == null)
    lastCreatedObj = firstObj;
  else
    lastCreatedObj = lastCreatedObj.nextObj;
  lastCreatedObj.created = true;
  return lastCreatedObj;
}
```

Figure 2.2: The `new` operator is assumed to have the semantics intended by this JAVA implementation manipulating the additional fields depicted in Figure 2.1.

On the set of types we define a strict order $\prec$ as follows. Since, in this thesis, we are interested in integer arithmetic, we define the type order only on the primitive JAVA types and our additional primitive types defined in Chapter 4. In Figure 2.3, an illustration of the following definition is given, where the arrows correspond to the relation symbol $\prec$.

Definition 2.2.2 (Strict order $\prec$ on primitive types)
The strict order $\prec$ *on the set*

$$T = \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{arithByte}, \texttt{arithShort}, \texttt{arithInt}, \texttt{arithLong}\}$$

of primitive types is defined as:

- $\texttt{byte} \prec \texttt{short}$,
- $\texttt{byte} \prec \texttt{arithByte}$,
- $\texttt{short} \prec \texttt{int}$,
- $\texttt{short} \prec \texttt{arithShort}$,
- $\texttt{int} \prec \texttt{long}$,
- $\texttt{int} \prec \texttt{arithInt}$, and
- $\texttt{long} \prec \texttt{arithLong}$.

Definition 2.2.3 (Transitive closure $\overset{*}{\prec}$)
The strict total order $\overset{*}{\prec}$ *denotes the transitive closure of* $\prec$.

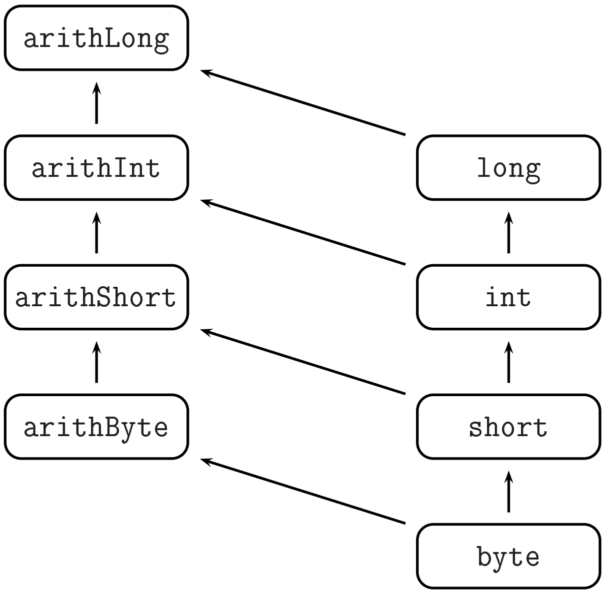


Figure 2.3: This figure illustrates the strict order $\prec$ on the set of primitive types. The arrows correspond to the relation symbol $\prec$.

The most important operator in a programming language is the assignment operator, especially when concerned with integer arithmetic. In Definition 2.2.4, we give a formal definition of the term *assignment compatibility* based on the strict total order $\stackrel{*}{\prec}$ on the set of primitive types. Intuitively, assignment compatibility means, that the type on the left-hand side of the assignment operator has a greater or equal range than the type of the right-hand side.

Definition 2.2.4 (Assignment compatibility)
Let $S, T \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{arithByte}, \texttt{arithShort}, \texttt{arithInt}, \texttt{arithLong}\}$ *be primitive types.*

S *is* assignment compatible *to* T *if and only if* S *and* T *are equal or* $S \stackrel{*}{\prec} T$.

Variables

In classical DL there is usually only one kind of variables which may occur in logical terms as well as in programs. But this is somehow misleading because a variable used in a program is actually not a variable but a non-rigid[2] 0-ary function. To

[2] Rigid functions have the same interpretation in each state of the model, whereas the interpretation of non-rigid functions may differ from state to state.

point out the difference, here we use two kinds of variables, as it is done in [Beckert, 2001].

The first kind is called *logical variable* which are denoted with $x, y, z, \ldots$. They are assigned the same values in all states and they have to be bound by a quantifier, this means free occurrences of logical variables are not allowed. Logical variables are allowed to be used in programs but only if a statement does not try to change the value of the variable. Thus, the statements x`=0;` and x`++;` are illegal. Logical variables may be instantiated with logical terms which of course only preserves the syntactical correctness of a formula but, in general, not its satisfiability or validity. Since DLJ is a typed logic, we use the notation $x \in \mathtt{T}$ to indicate that the variable x is of type $\mathtt{T}$ (see the example below).

The second kind of variables are the *program variables*, denoted by `x,y,z,...`. Program variables may have different values in different states but in one state a certain program variable always has same value. Thus, within a state a program variable may be seen as a constant. Program variables can be used in programs that may change their values. Actually, a program does not change the values of program variables but it changes the state and thus potentially the interpretation of the constant program variables. Finally, program variables cannot be quantified or instantiated with logical terms. The type of a program variable is defined by the declaration of the variable in the program it occurs.

In Example 2.2.5 the formula[3] (1.) is syntactically invalid because the statement x`=i;` is an assignment with a logical variable on the left-hand side. The reason why formula (2.) is not well-formed is, that the program variable `i` is illicitly quantified. Furthermore, the program variable `i` is used (in the illegal quantification) before it is declared and thus, the type of `i` is unknown. A syntactically correct formula with the intended meaning of formula (2.) is formula (3.).

Example 2.2.5
Examples of syntactically invalid formulas:

1. $\langle \mathtt{int\ i;}\ x\mathtt{=i;} \rangle x \doteq \mathtt{i}$
2. $\forall \mathtt{i} \langle \mathtt{int\ i=i;} \rangle \mathtt{i} \doteq \mathtt{i}$

An example of a syntactically well-formed formula:

3. $\langle \mathtt{int\ i;} \rangle \forall x \in \mathtt{int}\ (\{\mathtt{i} \leftarrow x\} \langle \mathtt{i=i;} \rangle \mathtt{i} \doteq \mathtt{i})$.

□

To conclude, what we call a logical variable really is a variable from the logical point of view. What we call a program variable is in fact not a variable but a non-rigid 0-ary function which may be interpreted differently in different states as it is usual in modal logics.

[3] The syntax of formulas is defined in definition 2.2.9

Signature

The signature Σ_{DLJ} of DLJ consists of a set of function and predicate symbols for each type of DLJ.

We do not give a formal definition of the whole signature (which would be too elaborate) but we only give a subset of the whole signature containing the symbols that will be used in the following.

Definition 2.2.6 (Subset of Σ_{DLJ} for the built-in types)
The signature Σ_{DLJ} contains for the built-in types $T = \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}\}$ the following function and predicate symbols. Most of the function symbols are the logical counterpart of the arithmetical JAVA *operators and, unless stated otherwise, their semantics is the same as that of the corresponding* JAVA *operator defined in Chapter 4.*

- *The binary function $JMul_T$ (corresponds to the* JAVA *operator* `*`*).*
- *The binary function $JDiv_T$ (corresponds to the* JAVA *operator* `/` *with the exception that the semantics of this function is not defined if the right-hand side argument is zero).*[4]
- *The binary function $JMod_T$ (corresponds to the* JAVA *operator* `%` *with the exception that the semantics of this function is not defined if the right-hand side argument is zero).*
- *The binary function $JPlus_T$ (corresponds to the* JAVA *operator* `+`*).*
- *The binary function $JMinus_T$ (corresponds to the* JAVA *operator* `-`*).*
- *The binary function $JShiftLeft_T$ (corresponds to the* JAVA *operator* `<<`*).*
- *The binary function $JShiftRight_T$ (corresponds to the* JAVA *operator* `>>`*).*
- *The binary function $JUnsigned\ ShiftRight_T$ (corresponds to the* JAVA *operator* `>>>`*).*
- *The binary function $JAnd_T$ (corresponds to the* JAVA *operator* `&`*).*
- *The binary function JOr_T (corresponds to the* JAVA *operator* `|`*).*
- *The binary function $JXor_T$ (corresponds to the* JAVA *operator* `^`*).*
- *The unary function JUnaryMinus (corresponds to the* JAVA *operator* `-`*).*
- *The unary function JComplement (corresponds to the* JAVA *operator* `~`*).*

[4] In JAVA, if the divisor is zero, the evaluation of the operator `/` terminates abruptly and an `ArithmeticException` is thrown.

- *The binary predicate symbols* $<, \leq, \doteq, \geq, >$ *(correspond to the* JAVA *operators* `<, <=, ==, >=, >`*).*
- *The unary function* $overflow_T$ *(for the semantics see Definition 2.2.14).*

Definition 2.2.7 (Subset of Σ_{DLJ} for the arithmetical types)
The signature Σ_{DLJ} *contains for the arithmetical types* $T = \{$`arithByte`, `arithShort`, `arithInt`, `arithLong`$\}$ *the following function and predicate symbols which have the usual mathematical semantics.*

- *The binary multiplication function* $*$.
- *The binary division function* $/$.
- *The binary remainder function mod.*
- *The binary addition function* $+$.
- *The binary subtraction function* $-$.
- *The unary plus and minus functions* $+$ *and* $-$.
- *The unary predicate* $inbounds_T$ *(for the semantics see Definition 2.2.13).*
- *The binary predicate symbols* $<, \leq, \doteq, \geq, >$.

As usual in mathematics, the semantics of the functions / and *mod* is not defined if the divisor is zero.

Finally, the signature Σ_{DLJ} contains the unary definedness predicate *isdef* whose semantics is defined in Definition 2.2.12

Terms

As usual, logical terms are constructed from the variables (both logical and program variables) and the constant and function symbols of all types.

In addition, certain JAVA expressions, that we will later call *simple left-hand sides* and *simple right-hand sides* (see Definition 6.2.1) are also logical terms.[5]

Definition 2.2.8 (Terms of DLJ)
The set $Term_{DLJ}$ *of terms of DLJ is defined as follows.*

- *Program variables and logical variables are terms.*

[5] Simple left-hand sides and simple right-hand sides are local variables, field and array accesses that have no side-effects which change their values. However, access to arrays or fields may have an exception thrown as a side-effect and in this case, the semantics of such term is undefined.

– *The constant and function symbols of all types are terms.*

– *The literals of the primitive* JAVA *types* `int` *and* `long` *are terms.*

– *String literals of* JAVA *are terms.*

– *The* `null` *literal of* JAVA *(which is of type* Null*) is a term.*

– *If* `o` *is a term of class type* `C` *and* `a` *is an field (attribute) of* `C`*, then* `o.a` *is a term.*

– *If* `a` *is a term of an array type and* `i` *is of type* `byte`, `short`, `int`, `arithByte`, `arithShort`, or `arithInt`*, then* `a[i]` *is a term.*[6]

Programs in DLJ

Programs contained in a DLJ formula consist of executable code and, as already mentioned, are not allowed to contain class or interface definitions. We do not require that these programs are compilable code even though they are extended by an according class definition. One reason why this requirement would be too rigorous, is that a program may be spread on several diamond operators $\langle\cdot\rangle$ (or box operators $[\cdot]$) as this is the case in formula (3.) in example 2.2.5. In fact, this problem could perhaps be solved by collecting all those program fragments but another reason is that we allow, under certain circumstances, programs in DLJ formulas (not in the context) to contain logical terms.[7]

A third reason why we do not require that programs contained in DLJ formulas have to compilable code is, that in our version of DL, JAVA programs may contain the additional arithmetical types. These types of course may not be contained in a program that are supposed to be compiled.

But at least, we require that the programs contained in DLJ formulas are well-typed. This means for example, that the type of the right-hand side of an assignment is assignment compatible to the left-hand side. For primitive types, this requirement is statically decidable, in contrast to reference types where runtime checks are necessary. The rules of our calculus are based on this assumption and thus, the soundness of the rules depends on it.

---

[6] According to the JAVA language specification [Bracha et al., 2000], variables of the type `long` are not allowed to be used as indices for array accesses.

[7] Of course, since logical variables are not allowed to be used in statements that try to change their values, this holds for logical terms as well. But everywhere in a program, instead of a JAVA expression of a certain type a logical term of the same type may be used if the evaluation of the expression does not try to change its value. Thus, the increment operator `++` may never be applied to a logical term, whereas a term on the right-hand side of an assignment is allowed.

Formulas

In definition 2.2.9, the set $Form_{DLJ}$ of DLJ formulas is inductively defined as usual in Dynamic Logic.

Definition 2.2.9 (Formulas of DLJ)
Relatively to a given DLJ signature Σ_{DLJ}, the set $Form_{DLJ}$ of DLJ formulas is inductively defined as follows.

- *true and false are DLJ formulas.*
- *If p is a predicate and $t_1, t_2, \ldots, t_n$ are terms of the appropriate types, then $p(t_1, t_2, \ldots, t_n)$ is a DLJ formula.*
- *If ϕ and ψ are DLJ formulas, then $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, and $\phi \rightarrow \psi$ are DLJ formulas.*
- *If ϕ is a DLJ formula and x is a logical variable, then $\forall x \phi$ and $\exists x \phi$ are DLJ formulas.*
- *If ϕ is a DLJ formula and p is a* JAVA *program (that satisfies the stipulations stated in the previous section), then $\langle p \rangle \phi$ and $[p]\phi$ are DLJ formulas.*
- *If ϕ is a DLJ formula and $\{\mathtt{i} \leftarrow x\}$ is an update (see Definition 2.2.10), then $\{\mathtt{i} \leftarrow x\}\phi$ is a DLJ formulas.*

As in [Beckert, 2001], we introduce so called *updates*, which may be attached to formulas to simplify notation.

Definition 2.2.10 (Update)
An update *is of the form $\{\mathtt{ls} \leftarrow t\}$, where* `ls` *is a program variable, a field access (i.e. is of the form* `o.a`*), or an array access (i.e. is of the form* `a[j]`*), and t is a logical term whose type is assignment compatible (see Definition 2.2.4) to the type of* `ls`*.*

The intended semantics of the update $\{\mathtt{ls} \leftarrow t\}$ attached to the formula ϕ is, that, before ϕ is evaluated in a state (see Definition 2.2.11), this state is changed by assigning `ls` the value of the term t. Thus, the semantics of $\phi^{\{\mathtt{ls} \leftarrow t\}}$ is the same as that of the formula $\langle \mathtt{ls} = t; \rangle \phi$.

Updates may also be concatenated. The semantics of $\phi^{(\{\mathtt{ls}_2 \leftarrow t_2\} \circ \{\mathtt{ls}_1 \leftarrow t_1\})}$ is the same as that of the formula $\langle \mathtt{ls}_1 = t_1;\ \ \mathtt{ls}_2 = t_2; \rangle \phi$.

2.2.3 Semantics

To define the semantics of DLJ, we first have to define the semantics of our program language which is as extension of JAVA. For the definition of the semantics of JAVA statements and expressions that do not involve integer arithmetic, we refer to the JAVA language specification [Bracha et al., 2000] and in case of doubt to the precise formal semantics of JAVA defined by Börger and Schulte [Börger and Schulte, 1999] using Abstract State Machines. The semantics of the (extended) JAVA integer arithmetic is defined in Chapter 4. The motivation for this semantics is given in Chapter 5.

Generally, as usual in modal logics, the models of DL are Kripke structures consisting of the set of possible worlds, a binary reachability relation on the set of worlds, and a usual first-order predicate logic structure. In the DL framework, worlds are called *states*. The reachability relation defines the possible state transitions caused by a program statement and thus strongly depends on the definition of the semantics of the underlying programming language.

To interpret function and predicate symbols, we use one fixed predicate logic model for all states. This means, that all functions (except the program variables which are actually non-rigid 0-ary functions) and predicates have the same meaning in all states. Furthermore, all states share the same universe consisting of an finite or infinite set of elements of all types.[8]

Logical variables are interpreted using a global variable assignment. As already mentioned, they have the same value in all states.

States

In classical DL, the set of states is usually defined as the set of variable assignments. But this is only possible, if the underlying programming language is simple (for example a while-language) and in particular not object-oriented. Since we use JAVA as a programming language, our definition of states is more involved.

Definition 2.2.11 (State)
A state *in DLJ consists of the assignments to*

- *the program variables (including* `this`*),*
- *the attributes (fields) of all objects (including arrays), and*
- *the class attributes (static fields) of all class types.*

[8] For example, the set of elements of the built-in types like `int` is finite, whereas the set of elements of the additional types like `arithInt` is infinite.

Note, that this definition of states contains no information about control flow (like a program counter) or the fact that an exception has been thrown. In contrast, Stenzel [Stenzel, 2001] defines an explicit store which contains references to the existing objects and arrays and information on the control flow.

Semantics of Terms, Programs, and Formulas

At first, we define the semantics of the function and predicate symbols of the signature Σ_{DLJ} defined in Section 2.2.2.

Simple left-hand sides and simple right-hand sides (see Definition 6.2.1) are defined such that their evaluation does not change their values. However, their evaluation is not free of side-effects since the evaluation of an array access or of an access to an attribute of an object may raise an `IndexOutOfBoundsException` or a `NullPointerException`. The rules of the calculus presented in Chapter 6 contain premisses whose validity guarantees the definedness of the access to arrays and attributes. For this purpose the unary predicate *isdef* is used whose semantics is defined in Definition 2.2.12.

Definition 2.2.12 (Unary predicate *isdef*)
Let simpleExpr be a simple left-hand side `slhs` *or a simple right-hand side* `srhs`*.*

The predicate isdef(simpleExpr) is true if and only if the evaluation of simpleExpr does not cause any exception.

To check whether the value of a logical term t of a built-in type or of an arithmetical type is in valid range (see Definition 5.1.1) of a certain type $\mathtt{T}$, the unary predicate $in_{\mathtt{T}}$ is defined.

Definition 2.2.13 (Unary predicate $in_{\mathtt{T}}$)
Let $\mathtt{T}$ *be a primitive integral built-in type or an arithmetical type and let* t *be a logical term of one these types. If* $\mathtt{T}$ *is an arithmetical type, let* $\mathtt{T}'$ *be the corresponding built-in type, otherwise* $\mathtt{T}'$ *is equal to* $\mathtt{T}$*.*

Then, the unary predicate $in_{\mathtt{T}}(t)$ *is defined as*

$$\mathtt{MIN\_}\mathtt{T}' \leq t \land t \leq \mathtt{MAX\_}\mathtt{T}'.$$

Expressions of the built-in types may cause overflow. In this case, according to the JAVA language specification, the value of the expression (after the computation of overflow) is defined by the n lowest order bits of the value (before the computation of overflow) in two's complement representation, where n is the number of bits used for storage of values of the type of the expression. In Definition 2.2.14, we define the unary function $overflow_{\mathtt{T}}$ which allows to compute overflow in decimal representation by means of the modulo function *mod* without converting the value in two's complement representation and vice versa.

Definition 2.2.14 (Unary function $overflow_T$)
Let T be one of the built-in types `byte`, `short`, `int`, *or* `long` *and let t be a logical term of one of these types.*

The function $overflow_T(t)$ is a short form for the term

$$(t - \texttt{MIN\_}T) \ mod \ (-2 * \texttt{MIN\_}T) + \texttt{MIN\_}T.$$

The above definition requires some explanation. Since the least values of the integral JAVA types is not zero, at first an according linear translation is performed by adding $(-\texttt{MIN\_}T)$. After the translation we have a range of values from zero to $\texttt{MAX\_}T - \texttt{MIN\_}T$ and now, we can apply the modulo function *mod* which has the usual mathematical meaning. Finally, we have to undo the linear translation by adding $\texttt{MIN\_}T$.

Now, that we have defined the semantics of the "special" functions and predicates contained in the signature Σ_{DLJ}, in the following we define the semantics of formulas.

As mentioned above, the definition of the semantics of a program p in DL is based on the binary reachability relation which is part of the Kripke structure. This relation defines the set of states reachable by the program p when started in a state s. Since JAVA is a deterministic programming language, this set is either empty (if p does not terminate) or contains exactly one state (if p terminates). We require, that the set of states is closed under the reachability relation. This means, that all states that may be reached by a program have to be contained in the set of states.

Definition 2.2.15 (Reachability relation)
Let s, t be states and p be a JAVA *program.*

The reachability relation $s\,[\![p]\!]\,t$ *is true if and only if the program p started in state s terminates in state t.*

The semantics of a term t contained in a program is the same as the semantics of a JAVA expression without side-effects that evaluates to the same value as t.

Before we can define the semantics of DLJ formulas, we first have to define the modification of variable assignments and states.

Definition 2.2.16 (Modification of a variable assignment)
Let β be a variable assignment, let x, y be logical variables, and let d be an element of the domain D. Then, the modification β_x^d of a variable assignment *is defined as follows.*

$$\beta_x^d(y) = \begin{cases} d & \textit{if } x = y \\ x & \textit{otherwise} \end{cases}$$

Definition 2.2.17 (Modification of a state)
Let $s \in S$ be a state, let $\mathtt{i}$ be a program variable, an attribute, a field, or a class attribute and let t be a logical term of a type that is assignment compatible to the type of $\mathtt{i}$.

The result of the modification $s^t_{\mathtt{i}}$ of the state s *is the state s' that differs from s only in the value of $\mathtt{i}$, which in s' is t.*

Definition 2.2.18 (Semantics of DLJ)
Let $\mathcal{K} = \langle S, [\![]\!], \mathcal{M} \rangle$ be a Kripke structure where S is the set of states, $[\![]\!]$ the binary reachability relation on the set of states, $\mathcal{M} = \langle D, I \rangle$ a first-order model, and β a variable assignment for the logical variables. Furthermore, let $val_{\mathcal{M},\beta}$ be a valuation function as usual for pure first-order formulas, let t be a logical term, $r, s \in S$ states, and $\phi, \psi \in Form_{DLJ}$ DLJ formulas. Then, the valuation function $val_{\mathcal{K},\beta} : S \times Form_{DLJ} \rightarrow \{\top, \bot\}$ is defined as follows.

$$val_{\mathcal{K},\beta}(s, \phi) = val_{\mathcal{M},\beta}(\phi) \quad \text{if } \phi \text{ is a pure first-order formula.}$$

$$val_{\mathcal{K},\beta}(s, \neg\phi) = \begin{cases} \top & \text{if } val_{\mathcal{K},\beta}(s, \phi) = \bot \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, \phi \wedge \psi) = \begin{cases} \top & \text{if } val_{\mathcal{K},\beta}(s, \phi) = \top \text{ and } val_{\mathcal{K},\beta}(s, \psi) = \top \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, \phi \vee \psi) = \begin{cases} \top & \text{if } val_{\mathcal{K},\beta}(s, \phi) = \top \text{ or } val_{\mathcal{K},\beta}(s, \psi) = \top \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, \phi \rightarrow \psi) = \begin{cases} \top & \text{if } val_{\mathcal{K},\beta}(s, \phi) = \bot \text{ or } val_{\mathcal{K},\beta}(s, \psi) = \top \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, \forall x\phi) = \begin{cases} \top & \text{if } val_{\mathcal{K},\beta^d_x}(s, \phi) = \top \text{ for all } d \in D \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, \exists x\phi) = \begin{cases} \top & \text{if } val_{\mathcal{K},\beta^d_x}(s, \phi) = \top \text{ for one } d \in D \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, \langle p \rangle\phi) = \begin{cases} \top & \text{if there is a state } r \text{ with } s\,[\![p]\!]\,r \text{ and} \\ & \text{it holds } val_{\mathcal{K},\beta}(r, \phi) = \top \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, [p]\phi) = \begin{cases} \top & \text{if for all states } r \text{ with } s\,[\![p]\!]\,r \text{ holds} \\ & val_{\mathcal{K},\beta}(r, \phi) = \top \\ \bot & \text{otherwise} \end{cases}$$

$$val_{\mathcal{K},\beta}(s, \{\mathtt{i} \leftarrow t\}\phi) = \begin{cases} \top & \text{if } val_{\mathcal{K},\beta}(s^t_{\mathtt{i}}, \phi) = \top \\ \bot & \text{otherwise} \end{cases}$$

Remark 2.2.19
Note, that the semantics for terms like `a[-5]`*, where* `a` *is of an array type, is not defined because in* JAVA*, the evaluation of the expression* `a[-5]` *does not yield a value but terminates abruptly and throws an* `IndexOutOfBoundsException`*. In our case, this means, that the semantics is underspecified and we assume, that such terms do not occur. In the rules of the calculus presented in Chapter 6, the validity of certain premisses, containing the definedness predicate isdef, guarantees that this assumptions holds. The same applies to terms of the form* `o.a`*.*

For us, this underspecified semantics is sufficient. However, if one wants to have a full specification of these terms, there are several possibilities. For example, one could define a special error element in the universe for such terms. This approach has the disadvantage, that terms of different types are equal if both are interpreted as the error element. To avoid this, one could define a different error element for each type, but then, for example, the terms `a[-5]` *and* `a[-6]` *are still equal.*

Definition 2.2.20 (Validity)
A DLJ formula ϕ is said to be true *in state $s \in S$ in the Kripke structure $\mathcal{K}$ if $val_{\mathcal{K},\beta}(s, \phi) = \top$ and we write $(\mathcal{K}, s) \models \phi$. Otherwise, ϕ is* false *in s and we write $(\mathcal{K}, s) \not\models \phi$.*

A DLJ formula ϕ is valid in the Kripke structure *$\mathcal{K}$ if ϕ is true in all states and we write $\mathcal{K} \models \phi$.*

A DLJ formula ϕ is valid *if ϕ is valid in all Kripke structures $\mathcal{K}$.*

Remark 2.2.21
In the following we will often use the term model *instead of (Kripke) structure.*

2.3 Sequent Calculi

In this section, we shortly introduce the principle of sequent calculi since the calculus presented in Chapter 6 is a sequent calculus.

2.3.1 Sequent

A sequent is of the form $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$ $(m, n \geq 0)$, where the ϕ_i and ψ_j are DLJ formulas. The formulas on the left-hand side of the sequent symbol $\vdash$ are called *antecedent* and the formulas on the right-hand side are called *succedent*.

The semantics of a sequent is the same as that of the formula $(\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$ $(m, n \geq 0)$.

A sequent $\phi_1, \ldots, \phi_m \vdash \psi_1, \ldots, \psi_n$ $(m, n \geq 0)$ is *valid* in a model $\mathcal{M}$ if and only if the formula $(\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)$ $(m, n \geq 0)$ is valid in $\mathcal{M}$.

2.3.2 Sequent Rule

Sequent rules are rule schemata of the form

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \Gamma_2 \vdash \Delta_2 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

where $\Gamma_i \vdash \Delta_i$ $(0 \leq i \leq n)$ and $\Gamma \vdash \Delta$ are sequents.

As usual, the sequents above the horizontal line in a rule schema are its premisses and the single sequent below is its conclusion.

Definition 2.3.1 (Soundness)
A sequent rule is sound *if from the validity of the premisses $\Gamma_i \vdash \Delta_i$ $(0 \leq i \leq n)$ follows the validity of the conclusion $\Gamma \vdash \Delta$.*

Note, that proof construction in sequent calculi in practice always starts with the original proof obligation and then, the rules are applied from bottom to top.

To refer to the calculus defined in Chapter 6 which extends the calculus defined in [Beckert, 2001], we define the symbol $\mathcal{C}_{DLJ}$.

Definition 2.3.2 (Calculus $\mathcal{C}_{DLJ}$)
The calculus $\mathcal{C}_{DLJ}$ *consists of the rules given in the sections 6.3 to 6.9 and the rules presented in [Beckert, 2001].*[9] *A sequent is* derivable *(with $\mathcal{C}_{DLJ}$) if it is an instance of the conclusion of a rule schema and all corresponding instances of the premisses of that rule schema are derivable sequents.*

2.4 Binary Representation of Primitive Integral Java Types

The programming language JAVA offers the five primitive integral data types `byte`, `short`, `int`, `long`, and `char`. These types have a finite range of values. Table 2.1 shows the ranges and the number of bits needed for storage of a value of each data type. In the following we will refer to the minimal and maximal value of a type *T* by `MIN_`*T* and `MAX_`*T*. Thus, for example the minimum value of type `byte` is denoted by `MIN_byte`.

The data type `char` is internally stored in unsigned binary representation. Although arithmetical operators can be applied on values and variables of that type, in the following we will disregard `char` because its values are interpreted as Unicode characters. Thus, we think this type is not suitable for arithmetic particularly as there are no negative values.

[9] The calculus given in [Beckert, 2001] contains rules for handling all JAVA constructs and also rules for handling exceptions.

data type	range	number of bits
`byte`	-128 to 127	8
`short`	-32768 to 32767	16
`int`	-2147483648 to 2147483647	32
`long`	-9223372036854775808 to 9223372036854775807	64
`char`	0 to 65535	16

Table 2.1: Primitive integral data types in JAVA.

The remaining four data types are stored in the so called *two's complement*. This representation is used in many programming languages because, in particular, additions and subtractions can be performed in a very efficient way. In the following, we will shortly explain the conversion from decimal representation into two's complement representation and vice versa.

We denote the bits in binary representation by $b_n, b_{n-1}, \ldots, b_0$ with b_0 as least significant bit and b_n as most significant bit which is used as sign. If the sign bit is set the number is negative, otherwise positive. In the following the number zero will be regarded as being positive.

Conversion from Decimal Representation into Two's Complement

First, we define the number n of bits we want to use for two's complement representation. If N is the number to be converted, then the unequation $n \geq 1 + \lceil ld(abs(N)) \rceil$ must hold.

Then, we have to check whether the number N to be converted is negative or positive.

If N is positive the sign bit b_n is set to zero and the remaining bits correspond to the usual unsigned binary representation. This means, the sum $\sum_{i=0}^{n-1} b_i * 2^i$ equals N.

On the other hand, if N is negative let M be the absolute value of N. Then we subtract one from M and proceed as in the case when N in positive. This yields the binary number B. The last step is to invert every single bit of B which, in particular, sets the sign bit to one.

Conversion from Two's Complement into Decimal Representation

Let n be again the number of bits used for two's complement representation.

Again, at first we have to check the sign bit of the binary number B to be converted. The sign is represented by the most significant bit b_n.

If the sign bit is set to zero the value of the number B in decimal representation is just the sum $\sum_{i=0}^{n-1} b_i * 2^i$.

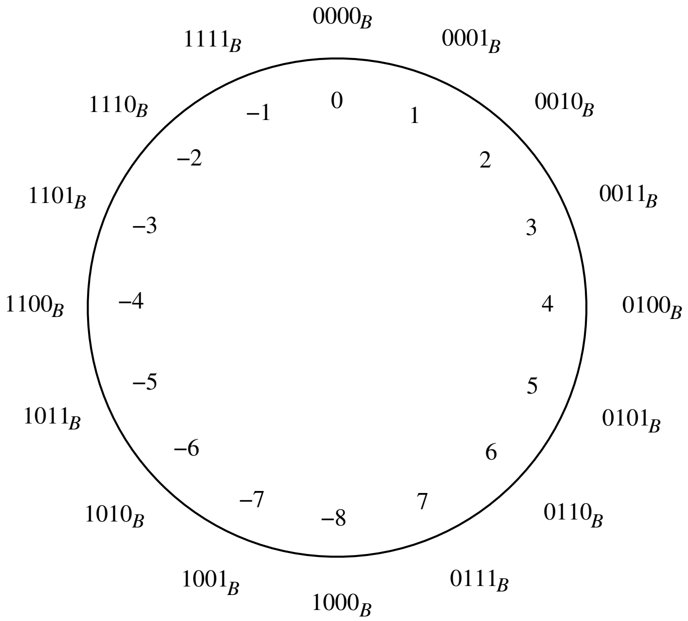


Figure 2.4: Two's complement representation of the number -8 to 7 with 4 bits

Otherwise, if the sign bit is set to one the value of B is $-(2^n)+\sum_{i=0}^{n-1} b_i * 2^i$. The value $-(2^n)$ is the smallest representable number in two's complement representation with n bits. The greatest representable positive number using n bits is 2^{n-1}.

As an example, Figure 2.4 shows the two's complement representation of the numbers -8 to 7 with 4 bits which is the minimal number of bits required.

2.4.1 Arithmetic in Two's Complement

As said before, the two's complement representation is often used because it allows performing arithmetical operations very efficiently. Especially, this applies to addition and subtraction because there is no need for taking care of the signs of the arguments. The sign bit is just handled as the remaining bits. As an example, we describe the addition of the numbers -2 and 5. As in Figure 2.4, we use 4 bits for two's complement representation and as the figure shows, the binary representation of -2 and 5 is 1110_B and 0101_B, respectively. Both numbers are added by adding bit by bit and considering carry which results in 0011_B, the two's complement representation of 3. Note, that for getting the right sign of the result, no special query was necessary.

3 Specification, Implementation, and Verification

There are two ways of constructing a software design.
One way is to make it so simple
that there are obviously no deficiencies.
And the other way is to make it so complicated
that there are no obvious deficiencies.
— C. A. R. HOARE

In this chapter we introduce the Unified Modeling Language (UML) which is used as modeling and specification language in the KeY project. The visual oriented UML contains as a subset the textual Object Constraint Language (OCL) which is more precise than the UML itself

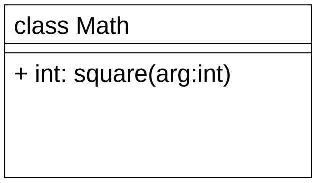
Then, we will introduce the target programming language JAVA CARD used in the KeY project and we will give reasons why we think this language is extremely suitable for program verification.

The last part of this chapter deals with the generation of proof obligations from the specification and implementation.

3.1 Specification

As already mentioned, we are using UML/OCL as specification language in the KeY project. The main reason for this choice is the fact that the UML is not only an OMG standard but also the industry standard for designing software systems. Especially the latter point is very important to us since the KeY project aims at integrating formal methods and program verification into the industrial software development process. It has turned out, that this goal cannot be achieved using very formal specification languages (like for example **Z** [Brien and Nicholls, 1992]) or artificial programming languages (used for example in the KIV system [Reif, 1995]). Another point is that there are many CASE tools available which support the UML. In the KeY project we decided to use the commercial CASE tool TOGETHER CONTROL CENTER from TogetherSoft because it is highly extensible and customisable.

Since the UML is a visual oriented and very universal modeling language, which has not been designed exclusively for modeling software systems, it is on its own




```
context Math::square(arg:Integer):Integer
post:  result=arg@pre*arg@pre
```

Figure 3.1: A simple UML model enriched with an OCL postcondition.

not precise enough for our needs. Fortunately, since version 1.3 the UML contains as a subset the OCL which allows to add formal constraints to an UML model in order to make it more precise. Constraints in that language are always understood in the context of a UML model, they never stand alone. For example, the OCL allows to attach pre- and postconditions to operations[1] and offers the specification of invariants.

An example of a simple UML model enriched with the OCL is depicted in Figure 3.1. The UML diagram consists of the single class `Math` which has an operation `square` that is supposed to calculate the square of the argument. This property, which cannot be expressed in the UML, is specified by the attached OCL postcondition. The first line of the constraint specifies the context of the postcondition that in this case is the operation `square` within the class `Math`. The second line specifies the property that must hold after the execution of the operation. The variable `result` used in this constraint is a reserved variable of the OCL which always refers to the return value of a non-void operation. Another built-in feature of the OCL is the `@pre` operator which may be attached to an attribute which then refers to its value at the beginning of the operation.

3.1.1 Semantics of the OCL

A crucial point of the OCL is that there is no official rigorous semantics. Even the OMG standard document [OMG, 1999a] is not precise enough. A possibility to give a precise semantics is to translate the OCL into a known, well understood language or logic. Such a translation can be found in [Keller, 2002] where the OCL is translated into Dynamic Logic. In this thesis, we will adopt this approach which is also used in the KeY project.

OCL expressions only make sense in the context of a UML model. The signature Σ_{OCL} of the Dynamic Logic which is the target of the translation also depends on

[1] In the context of object-oriented programming operations are called *methods*.

the given UML model but always contains a rigid core, namely the signature of the OCL itself. Especially, Σ_{OCL} always contains the arithmetical functions and predicates of the OCL, denoted by Σ_{OCL}^{arith}.

Definition 3.1.1 (Arithmetical Signature of the OCL)
The arithmetical signature Σ_{OCL}^{arith} *of the* OCL *consists of*

- *the function symbols* $*, /, +, -$ *and*
- *the predicate symbols* $<, \leq, =, \geq, >$.

For evaluating the DL formulas resulting from the translation we use a canonical model $\mathcal{M}_{OCL} = \langle \mathbb{Z}, I_{OCL} \rangle$ which assigns all the function and predicate symbols of Σ_{OCL} the meaning defined in the OCL specification [OMG, 1999a]. Of course, it is not easy to verify that the model really corresponds to the definitions in the specification. But we are only interested in a subset $\mathcal{M}_{OCL}^{arith}$ of the whole model assigning the function and predicate symbols in Σ_{OCL}^{arith} the intended meaning.

Definition 3.1.2 (Canonical arithmetical model)
The canonical arithmetical model $\mathcal{M}_{OCL}^{arith} = \langle \mathbb{Z}, I_{\mathbb{Z}} \rangle$ *consists of*

- *the domain* $\mathbb{Z}$ *which denotes the set of integers (as usual in mathematics) and*
- *the interpretation function* $I_{\mathbb{Z}}$ *which assigns the symbols in* Σ_{OCL}^{arith} *their usual mathematical meaning.*

In the OCL specification it is explicitly stated that the OCL type INTEGER[2] represents the mathematical concept of integers and that the operators $*, /, +,$ and $-$ have the usual mathematical semantics. Thus, in contrast to the whole canonical model, it is easy to see that $\mathcal{M}_{OCL}^{arith}$ really corresponds to the OCL specification.

At this time we do not say anything about the underlying programming language and its semantics. In Section 5 we will discuss three different semantics for JAVA integer arithmetic which lead to different notions of "correctness".

3.2 Implementation

As mentioned in Chapter 1, the KeY project aims at integrating formal methods into the industrial software development process and therefore the subset JAVA CARD of JAVA was chosen as target programming language since it is a real modern language. JAVA CARD does not offer the JAVA data types `int` and `long` but nevertheless, in the following we will not only deal with the data types `byte` and `short`. The calculus

[2] The OCL translation directly maps the OCL type INTEGER to the set $\mathbb{Z}$ of integers.

```
public int square(int arg) {
  int res=0;
  if (arg<0) arg=-arg;
  for (int i=0; i<arg; i++)
    res=res+arg;
  return res;
}
```

Figure 3.2: A possible implementation of the square method specified in Section 3.1

presented in Chapter 6 also contains rules for the data types `int` and `long` because there is no substantial difference concerning our work between handling the latter and the former. Also, giving rules for all the four data types enables us to use the calculus for full JAVA integer arithmetic as well. Thus, from the next section on we will talk about JAVA integer arithmetic but all considerations apply to the integer data types of JAVA CARD as well.

In the previous section we have given a small example of a UML/OCL specification. In Figure 3.2 we present a possible implementation for this sample specification in JAVA. To make the implementation a little bit more complicated, the square of the argument is not calculated by just using the multiplication operator but by repeated addition.

In the next section we will explain how specification and implementation yield the starting point for verification.

3.3 Verification

Program verification means proving the correctness of a program with regard to a given specification. This is done by proving certain conditions that are generated from the specification and the implementation. In the KeY project, these conditions are formulas of first-order Dynamic Logic (DL) [Harel, 1979]. The definition of syntax and semantics of our version of DL, called DLJ, can be found in Section 2.2.

A Dynamic Logic[3] can be seen as a multi-modal predicate logic with modalities $\langle p \rangle$ for each legal program p of the target programming language the DL is based on. A modality $\langle p \rangle$ (read "diamond p") refers to the successor worlds (called "states" in the

[3] We are knowingly talking about "a Dynamic Logic" because depending on the programming language one gets several variants of Dynamic Logic with different properties.

DL framework) reachable by running the program p. In general, there can be several successor states because the associated programming language does not have to be deterministic. In this paper we use Java which is a deterministic programming language. This means, that a modality $\langle p\rangle$ has exactly one successor state (if p terminates) or none (if p does not terminate). A DL-formula $\langle p\rangle\psi$ expresses that p terminates in a state that satisfies the formula ψ. In DL there still are other modal operators $[p]$ (read "box p") for each program p. A formula $[p]\psi$ requires all states, in which p terminates, to satisfy ψ. Basically, one could leave out either the modalities $\langle p\rangle$ or $[p]$ without losing expressiveness because the formula $\neg\langle p\rangle\neg\psi \leftrightarrow [p]\psi$ is a tautology and thus, any formula can be transformed in an semantically equivalent formula containing either only box or diamond modalities.

DL can also be seen as a kind of Hoare logic. The formula $\phi \rightarrow [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, DL is closed under the logical operators. This means that the formulas ϕ and ψ may itself contain programs enclosed by modal operators whereas in Hoare logic they have to be pure first-order predicate logic formulas.

Given a DL based on a deterministic programming language, the diamond operator expresses total correctness, which means that the program terminates and is correct. On the other hand, the box operator expresses partial correctness. This means that the program possibly not terminates but if it terminates, then it is correct.

We believe that DL is suitable for program verification because deduction in DL is based on symbolic program transformation which comes close to the programmer's understanding of how a program works. Proving in DL with an interactive theorem prover is similar to using a symbolic debugger which likewise symbolically executes statement by statement of the program's source code and offers the possibility to see the assignments of the variables.

As mentioned above, a program is said to be correct if certain conditions generated from the specification and implementation hold. These conditions are in our case DL formulas. Thus, verifying the correctness of a program with respect to a specification means proving the validity of certain DL formulas. But now the question comes up where these DL formulas come from or how they are generated. Answering this question is not a topic of this thesis and we refer to [Keller, 2002]. He presents a method for translating a UML/OCL specification and an implementation into DL formulas. This translation is very technical and we will not dwell on it. But to give at least an idea of how such a translation works we will give a simple example. Remember the sample specification depicted in Figure 3.1 and the possible implementation shown in Figure 3.2. In Figure 3.3, one can see the resulting DL formula generated by the translation described in [Keller, 2002]. In the following we will call such generated DL formulas "proof obligations". As one would expect, the program within the diamond operator is the implementation of the method `square` which the OCL postcondition refers to. The formula supposed to hold after the execution of the program is just the OCL postcondition where the reserved OCL variable `result` was replaced by the actual return variable `res`.

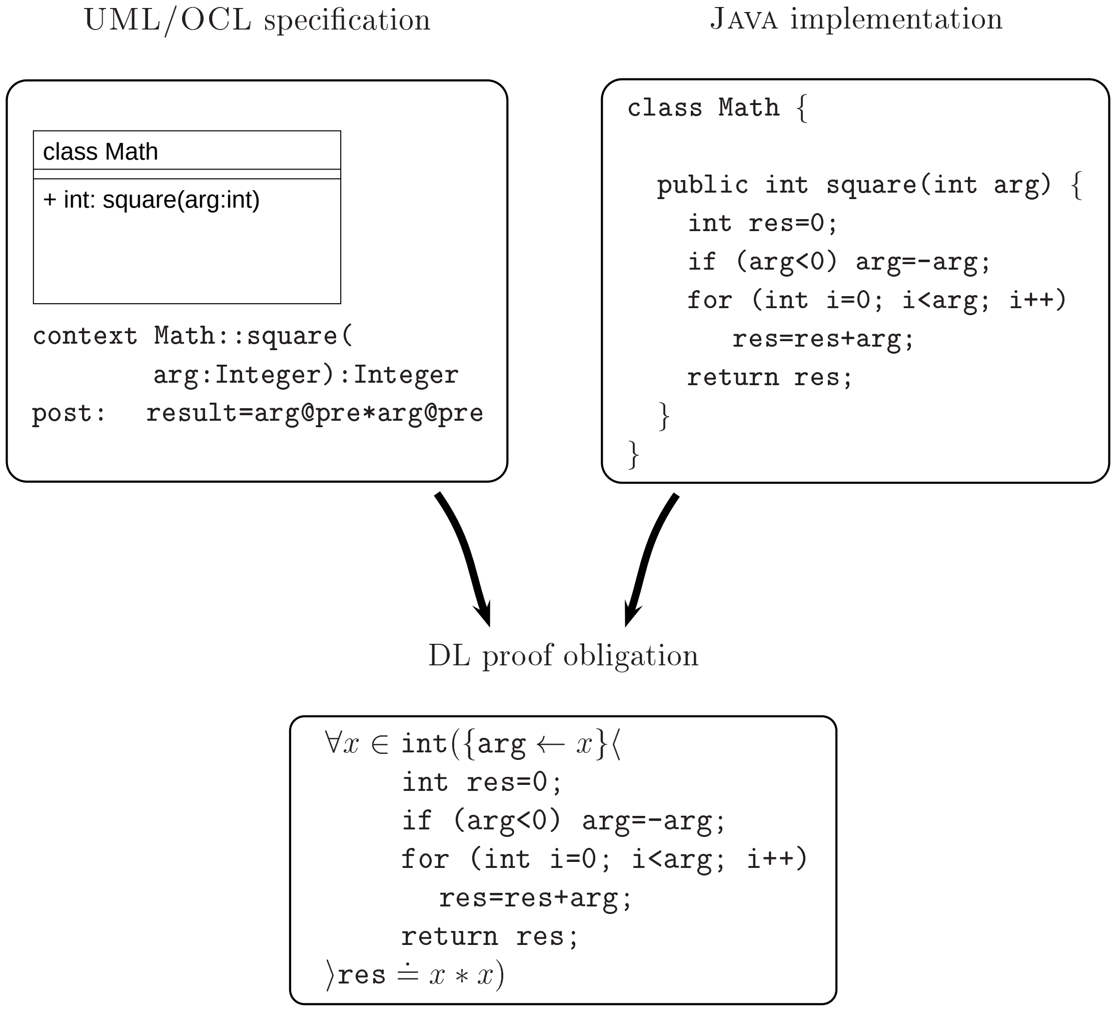


Figure 3.3: A simple example showing how a proof obligation is generated from an UML/OCL specification.

3.3.1 Semantical Differences between Specification and Implementation

Specification languages usually are very much mathematically oriented because they should be as precise as possible. But this also means that they are often very conceptional and abstract away from technical details concerning a certain implementation language. On the one hand this is necessary because during specification one does not want to be concerned with too many details but on the other hand there are some problems arising from that abstraction. As an example, in the following, we will regard problems concerning integral data types and operations on them.

We will restrict ourselves to the specification language UML/OCL and to the implementation language Java.

For specifying propositions about integer numbers in UML/OCL there is the basic data type Integer. This data type is isomorphic to the integer numbers $\mathbb{Z}$ and thus has an infinite range. For operations on the data type Integer there are the usual operators for addition, multiplication, and so on. They have the usual mathematical semantics.

On the other hand in Java there are the four integral primitive data types `byte`, `short`, `int`, and `long` which all have a different but finite range. There is still another integral data type `char` which we will not consider in the following because values assigned to a variable of this type are interpreted as Unicode characters. In Java, there are the same operators as in OCL to perform arithmetic on these data types. But in Java the semantics of the arithmetical operators differs from the mathematical semantics they have in OCL. The main reason for the different semantics is of course the finiteness of the data types. The Java virtual machine performs arithmetical operations in the binary system. If the result of an operations exceeds the fixed number of bits reserved for storage then the surplus high-order bits are disregarded. This process is called overflow or underflow, respectively, whichever the result exceeds the upper or the lower bound. The consequence of overflow and underflow[4] of course is that the result of an operation performed in Java does not always coincide with the mathematical result of the operation.

3.4 Correctness

In this section we dwell on problems with the definition of the term "correctness". Actually, it is clearly defined what correctness means, namely that a program meets its specification. But problems arise from the differences in the semantics of OCL and Java. Concerning our work we have pointed out two differences in the previous section. The first difference is that in the specification a infinite integer data type is used which is then implemented by finite data types. The second concerns the semantics of the operators which is a consequence of the type discrepancy.

As mentioned in Section 3.3, in the KeY project DL is used as the logical basis for program verification. The correctness of a program is established by proving the validity of DL formulas, the proof obligations, in the canonical model. Thus, the definition of the semantics of DL and especially the definition of the underlying programming language determines which programs are called "correct" and which

[4] From now on, when talking about overflow the same also applies to underflow unless explicitly stated otherwise.

are not. In chapter 5, we will give three different DL semantics for Java integer arithmetic and discuss their effects on correctness.

4 Extension of Syntax and Semantics of Java

We all agree that your theory is crazy,
but is it crazy enough?

— NIELS BOHR

In this chapter we present our extensions of the JAVA syntax with additional types and define the semantics $\mathcal{S}_{KeY}$ of these types and the operators acting on them. The motivation for the extensions and the semantics $\mathcal{S}_{KeY}$ is given in Chapter 5. Perhaps, the reader may first read the motivation and then continue with this chapter. The reason, why we first define the extensions of syntax and semantics is, that the notions defined in this chapter are needed to explain the motivation.

The semantics $\mathcal{S}_{KeY}$ is an operational semantics and we give the definitions in an natural language style as it is done in the JAVA language specification [Bracha et al., 2000] but we try to be as precise as possible. Note, that the arithmetical data types, introduced in the following, and the operators acting on them, seen as abstract data types, are no refinement of the built-in JAVA data types.

4.1 Extension of Java Syntax

In this section, we define the extension of JAVA syntax by additional primitive types and methods.

4.1.1 Additional Primitive Types

In addition to the built-in primitive integral JAVA types `byte`, `short`, `int`, and `long`, we introduce the four additional primitive integral types `arithByte`, `arithShort`, `arithInt`, and `arithLong`. In contrast to the built-in types, the additional types are called *arithmetical types*.

4.1.2 Additional Methods

In this section we extend the built-in JAVA class `java.lang.Object` by additional methods. Since the class `Object` is the root of the inheritance tree, each class inherits

```
final arithInt overflow(arithInt arg, String op)
final arithInt overflow(arithInt arg1, arithInt arg2, String op)
final arithLong overflow(arithLong arg, String op)
final arithLong overflow(arithLong arg1, arithLong arg2, String op)
```

Figure 4.1: This figure shows the signature of the additional methods in class `Object`.

these additional methods. Note, that these methods cannot be overwritten in the subclasses of `Object` since they are declared as final. Figure 4.1 shows the signature of the additional methods.

In contrast to the built-in methods of class `Object` like `toString`, the implementation and thus, the semantics, of the additional methods `overflow` is unspecified. In Chapter 5, we will explain what they are needed for.

Furthermore, we define, that the methods `overflow` are not allowed to be invoked explicitly be a method call statement. Rather, as will be explained later, they are implicitly called by the JAVA virtual machine if overflow occurs in operations of the arithmetical types.

4.2 Extension of Java Semantics

In this section we define the semantics of the additional arithmetical types and the operational semantics $\mathcal{S}_{KeY}$ of the operators acting on them.

4.2.1 Range of the Arithmetical Types

The arithmetical types all have an infinite range and thus, they are isomorphic to the OCL type INTEGER.

We do not define literals for the arithmetical types but literals of the built-in types `int` and `long` may be assigned to variables of the arithmetical types `arithInt` and `arithLong`, respectively.[1] Since there are no literals of the arithmetical types (which could be arbitrarily big since these types have an infinite range) it is not possible to assign a value to a variable of an arithmetical type which is not representable in the corresponding built-in type.

[1] According to the JAVA language specification, there are no integer literals of the types `byte` and `short`.

4.2.2 Operations on the Arithmetical Types

The JAVA operators that act on the primitive built-in integer and boolean types may also be used on the additional arithmetical types. Here, we just give an overview about the set of operators; their exact semantics is defined in Section 4.2.6. The precedence of the operators, according to the JAVA language specification, is depicted in Table 4.1.

The following operators act on the primitive built-in types and may also be used for the additional arithmetical types, unless stated otherwise:

- The comparison operators which result in a value of type `boolean`:
  - The numerical comparison operators `<`,`<=`,`>`, and `>=`.
  - The numerical equality operators `==` and `!=`.
- The numerical operators which result in a value of type `int`, `long`, `arithInt`, or `arithLong`:
  - The unary plus and minus operators `+` and `-`.
  - The multiplicative operators `*`,`/`, and `%`.
  - The additive operators `+` and `-`.
  - The prefix increment and decrement operator `++` and `--`.
  - The postfix increment and decrement operator `++` and `--`.
  - The signed and unsigned shift operators `<<`,`>>`, and `>>>`.
    These operators are not allowed for arithmetical types (see Section 4.2.6).
  - The bitwise complement operator `~`.
    This operator ist not allowed for arithmetical types (see Section 4.2.6).
  - The bitwise integer and logical operators `&`,`|`, and `^`.
    These operators are not allowed for arithmetical types (see Section 4.2.6).
  - The conditional operator `? :`.
  - The cast operator which can convert from any integral value to a value of any specified numeric type.
  - The string concatenation operator `+`.
    This operator is not allowed for arithmetical types (see Section 4.2.6).
- Other operators:
  - The logical complement operator `!`.
  - The boolean logical operators `&`, `|`, and `^`.

Group	Operator
postfix operators	`++ --`
unary operators	`++ -- + - ! ~`
multiplicative operators	`* / %`
additive operators	`+ -`
shift operators	`<< >> >>>`
relational operators	`< > <= >=`
equality operators	`== !=`
bitwise "and" operator	`&`
bitwise "xor" operator	`^`
bitwise "or" operator	`|`
logical "and" operator	`&&`
logical "or" operator	`||`
conditional operator	`? :`
assignment operators	`= *= /= %= += -= &=` `^= |= <<= >>= >>>=`

Table 4.1: This table shows the precendence of the JAVA operators.

- The logical conditional operators `&&` and `||`.
- The simple and compound assignment operators `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, and `>>>=`.

  The compound assigment operators `&=`, `^=`, `|=`, `<<=`, `>>=`, and `>>>=` containing bit operators are not allowed for arithmetical types (see Section 4.2.6).

4.2.3 Numeric Promotion

In this section, we define *numeric promotion* which is needed for the definition of the semantics of the arithmetical operators.

The semantics of JAVA operators is only defined for the types `int` and `long`. Thus, before an operation can be performed, the types of the operands must be converted into the types the operation is defined on. This process is called *numeric promotion*.

The type conversion performed by numeric promotion never yields a type with a smaller range but either leaves the type unchanged (*identity primitve conversion*) or yields a type with a greater range (*widening primitve conversion*). There also exists a conversion called *narrowing primitve conversion* which yields a type with a smaller range but this kind of conversion can only occur in explicit type casts.

According to the number of arguments of the operators *unary numeric promotion* and *binary numeric promotion* are distinguished.

Unary Numeric Promotion

Some operators perform unary numeric promotion to a single operand before actually performing the operation. According to the JAVA language specification, the operand must be a primitive numeric type otherwise a compile-time error occurs. Additionally, we define unary numeric promotion for the arithmetical types. The following rules apply in order and result in the promoted type which is a numeric type as well.

1. If the operand is of type `byte` or `short`, the operand is converted to `int`.
2. Otherwise, if the operand is of type `arithByte` or `arithShort`, the operand is converted to `arithInt`.
3. Otherwise, the type of the operand remains as is.

Binary Numeric Promotion

If binary numeric promotion is applied, each operand must be of a primitive integral built-in type or of a arithmetical type.[2]

When binary numeric promotion is performed, the following rules apply in order.

1. If both operands are of built–in types, the following rules apply in order.
   - If either operand is of type `long`, the other is converted to `long`.
   - Otherwise, both operands are converted to `int`.
2. Otherwise, if both operands are of arithmetical types, the following rules apply in order.
   - If either operand is of type `arithLong`, the other is converted to `arithLong`.

[2] In the JAVA language specification, binary numeric promotion is also defined for the floating point types `float` and `double`. Since we are only concerned with integer arithmetic, we disregard these types and assume that neither of the operands is of a floating point type.

- Otherwise, both operands are converted to `arithInt`.

3. Otherwise, if the types of the operands are mixed, this means that either is a built–in type and the other is an arithmetical type, the following rules apply in order.

   - If either operand is of type `arithLong`, the other is converted to `arithLong`.
   - Otherwise, if either operand is of type `long`, both operands are converted to `arithLong`.
   - Otherwise, both operands are converted to `arithInt`.

4.2.4 Conversion

The semantics for the arithmetical operators is only defined for the types `int`, `long`, `arithInt`, `arithLong`. But as a convenience for the programmer, explicit casts of the types of the operands into these types are only needed seldom. Mostly, these casts are performed implicitly by the virtual machine. The process of changing the type of an expression implicitly but also explicitly (by means of applying a type cast operator) is called *conversion*.

4.2.5 Kinds of Conversion

Specific type conversions in JAVA are divided into seven categories. Concerning our work, only the following four categories are of interest.

Identity Conversion

The identity conversion is a conversion from any type to that same type and is allowed for all arithmetical types as well.

This may seem trivial, but it allows a more homogenous definition of the semantics of the operators because, for example, it can be defined that each operand expression is subjected to conversion without stating an exception for the case that the operand already has the required type.

Widening Primitive Conversion

The following conversions are called *widening primitve conversions* since the type after conversion has a greater range than the type before.

- `byte` to `short`, `int`, `long`, `arithByte`, `arithShort`, `arithInt`, or `arithLong`.
- `short` to `int`, `long`, `arithShort`, `arithInt`, or `arithLong`.

– `int` to `long`, `arithInt` or `arithLong`.

– `long` to `arithLong`.

– `arithByte` to `arithShort`, `arithInt`, or `arithLong`.

– `arithShort` to `arithInt` or `arithLong`.

– `arithInt` to `arithLong`.

These widening primitive conversions do not lose any information. This means they do not change the values of the operands.

Using the strict total type order $\overset{*}{\prec}$ (see Definition 2.2.3), widening primitive conversion can be defined more formally as follows.

Definition 4.2.1 (Widening primitve conversion (formalised))
Let $T = \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{arithByte}, \texttt{arithShort}, \texttt{arithInt}, \texttt{arithLong}\}$ *be the set of primitive built-in and arithmetical types, let* $R, S \in T$*, and let* $(\overset{*}{\prec}, T)$ *be the strict total order on* T *(see Definition 2.2.3).*

A conversion from type R *to type* S *is called a* widening primitive conversion *if and only if* $R \overset{*}{\prec} S$.

Narrowing Primitive Conversion

The following conversions are called *norrowing primitve conversions* since the type after conversion has a smaller range than the type before.

– `short` to `byte`.

– `int` to `short` or `byte`.

– `long` to `int`, `short`, or `byte`.

– `arithByte` to `byte`.

– `arithShort` to `arithByte`, `short`, or `byte`.

– `arithInt` to `arithShort`, `arithByte`, `int`, `short`, or `byte`.

– `arithLong` to `arithInt`, `arithShort`, `arithByte`, `long`, `int`, `short`, or `byte`.

Using the strict total type order $\overset{*}{\prec}$ (see Definition 2.2.3), narrowing primitive conversion can be defined more formally as follows.

Definition 4.2.2 (Narrowing primitve conversion (formalised))
Let $T = \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{arithByte}, \texttt{arithShort}, \texttt{arithInt}, \texttt{arithLong}\}$ *be the set of primitive built-in and arithmetical types, let* $R, S \in T$*, and let* $(\overset{*}{\prec}, T)$ *be the strict total order on* T *(see Definition 2.2.3).*

A conversion from type R *to type* S *is called a* narrowing primitive conversion *if and only if* $S \overset{*}{\prec} R$*.*

If the type to be converted is a built-in JAVA type, narrowing conversion may change the value of the expression, whose type is to be converted, if it is not representable in the target type. In this case overflow occurs and the value of the expression after conversion are the n lowest order bits of the expression in two's complement representation, where n is the number of bits used for storage of values of the target type.

On the other hand, if overflow occurs in a narrowing conversion from an arithmetical type to another arithmetical type or to a built-in type, the method `overflow` is invoked, as will be explained in the following sections.

4.2.6 Semantics of Operators

In the following, we define the semantics of the arithmetical JAVA operators. Since we are not concerned with floating point arithmetic, we assume, that the operands of the operators are of primitive integral built-in types or of the arithmetical types.

Numerical Comparison Operators <,<=,>, and >=

First, binary numeric promotion is applied to the operands and then signed integer comparision, as usual in mathematics, is performed. The type of a comparision expression is the primitive type `boolean`.

Numerical Equality Operators == and !=

Also, first binary numeric promotion is performed on the operands and then the values are compared as usual in mathematics. The type of an equality expression is `boolean`. The equality operators are of lower precedence than the comparision operators.

Unary Plus and Minus Operators + and -

Unary numeric promotion is applied to the operand and the type of an unary plus or minus expression is the promoted type of the operand.

The result of a unary plus expression is equal to the value of the operand.

To evaluate the result of an unary minus expression, we have to make the following case distinction:

- If the promoted type of the operand is `int` or `long` the result is the mathematical negation of the operand except for the following two cases. If the value of the operand is `MIN_int` or `MIN_long`, respectively, the value of the unary minus expression is, due to overflow, the same as the value of the operand itself. Thus, the expression `MIN_int == -MIN_int` evaluates to `true`.
- If the promoted type of the operand is `arithInt` or `arithLong` the unary minus operator has the same semantics as in mathematics and thus, the result of an unary minus expression is the mathematical negation of the operand value except for the following two cases. If the value of the operand *arg* is `MIN_int` or `MIN_long`, respectively, the result is the return value of the method invocation `overflow(`*arg*`,"-")`. In this case, the method `overflow` is invoked because the values `MIN_int` and `MIN_long`, respectively, are in valid range (see Definition 5.1.1), but the negation of these values is not.

The reason for this "strange" semantics is due to the fact that JAVA internally stores numbers in so called two's complement representation (see Section 2.4). Typical for two's complement representation is that there is always one negative number more than there are positive numbers. Thus, the (mathematical) negation of the smallest number is not representable because it is by one greater than the greatest representable positive number.

Multiplicative Operators `*`, `/`, and `%`

The first step in the evaluation of all the three operators is to apply binary numeric promotion to the operands. The type of a multiplicative expression is the promoted type of the operands.

- **Multiplication operator `*`**

  If the promoted type is `int` or `long`, then, if no overflow occurs, the result is the mathematical product of the operands. If overflow occurs, the result is the decimal representation of the 32 (if the promoted type is `int`) or 64 (if the promoted type is `long`) lowest order bits of the mathematical product, represented in some sufficiently large two's complement represenation. Note, that if overflow occurs, even the sign of the JAVA product may differ from the sign of the mathematical product.

  If the promoted type is `arithInt` or `arithLong`, two cases have to be distinguished.

If both arguments arg_1 and arg_2 are in valid range (see Definition 5.1.1), then the result is the mathematical product if it is in valid range, otherwise the result is the return value of the method invocation `overflow(`arg_1`,`arg_2`,"*")`.

If at least one argument is not in valid range, then the result is the mathematical product of the operands and method `overflow` is not called.

- **Division operator /**

  The operand on the left-hand side is called *divident* and the operand on the right-hand side is called *divisor*. If the divisor is 0, an `ArithmeticException` is thrown and the evaluation of the division expression terminates abruptly.

  If the promoted type of the operands is `int` or `long`, then, if no overflow occurs, the result is the mathematical integer quotient of the divident and divisor. Overflow may only occur in just one case, namely if the value of the divident is equal to `MIN_`T (where T is the promoted type of the arguments) and the value of the divisor is -1. Then, the result of the division expression is, due to overflow, `MIN_`T. This case is similar to applying the unary minus operator to `MIN_`T (see above).

  If the promoted type of the operands is `arithInt` or `arithLong`, two cases have to be distinguished.

  If both arguments arg_1 and arg_2 are in valid range (see Definition 5.1.1), then the result of the division operation is the mathematical integer quotient if it is in valid range. Otherwise the result is the return value of the method invocation `overflow(`arg_1`,`arg_2`,"/")`. The only case, when this may happen, is when the value of the divident is `MIN_`T (where T is the corresponding primitve JAVA type to the promoted type `arithInt` or `arithLong`) and the value of the divisor is -1.

  If at least one argument is not in valid range, then the result is the mathematical integer quotient of divident and divisor and method `overflow` is not invoked.

- **Remainder operator %**

  The remainder operator `%` yields the remainder of the operands from an implied division. The operand on the left-hand side is the divident and the operand on the right-hand side is the divisor. If the divisor is 0, an `ArithmeticException` is thrown and the evaluation of the remainder expression terminates abruptly.

  In contrast to the other multiplicative operators `*` and `/`, the result of an remainder operation is always strictly less than the divident and thus, no overflow may occur. Therefore, the evaluation of an remainder operation never causes the invocation of the method `overflow`.

  If `a` is the divident and `b` the divisor, then the remainder operator always produces a result value such that `(a/b)*b+(a%b)` is equal to `a`, no matter whether the promoted type of the operands is `int`, `long`, `arithInt`, or `arithLong`. This identity even holds in the special case that the divident is `MIN_`T and the divisor is -1 (in this case, the remainder is 0).

Additive Operators + and -

For both operators, first binary numeric promotion is applied to the operands. The type of an additive expression is the promoted type of the operands.

If the promoted type of the operands is `int` or `long`, then, if no overflow occurs, the result is the mathematical sum or difference, respectively, of the operands. If overflow occurs, the result is the decimal representation of the 32 (if the promoted type is `int`) or 64 (if the promoted type is `long`) lowest order bits of the mathematical sum or difference, respectively, represented in some sufficiently large two's complement represenation. Note, that if overflow occurs, the sign of the addivitve JAVA expression differs from the sign of the mathematical result.

If the promoted type of the operands is `arithInt` or `arithLong`, then, two cases have to be distinguished.

If both operands arg_1 and arg_2 are in valid range, then the result is the mathematical sum or difference, respectively, if it is in valid range. Otherwise the result is the return value of the method invocation `overflow(`arg_1`,`arg_2`,"+")` or `overflow(`arg_1`,`arg_2`,"-")`, respectively.

If at least one operands is not in valid range, then the result is the mathematical sum or difference, respectively, and method `overflow` is not invoked.

Prefix Operators ++ and --

An unary expression preceded by the prefix operator `++` (`--`) is called a prefix increment (decrement) expression, respectively. Note, that the prefix operators can only be applied to an expression that evaluates to a variable, otherwise a compile-time error occurs. The type of a prefix expression is the type of the operand and *not* the promoted type of the operand.

The intended semantics of a prefix operator is to increment (decrement) the value of the operand variable by one. The value of a prefix increment (decrement) expession is the value *after* the increment (decrement). Especially, this is important if the increment (decrement) expression is part of a compound expression.

Evaluating a prefix increment (decrement) expression, the operand is increased (decreased) by one using the JAVA addition (subtraction) operator with the operand and the integer literal `1` as arguments. If necessary, the sum (difference) is narrowed by a primitive narrowing conversion to the type of the variable before it is stored. Thus, the expression `++a` is semantically equivalent to `(`T`)(a+1)` where T is the promoted type of `a` and the integer literal `1`.

Since prefix expressions are evaluated by internally performing a JAVA addition or subtraction, respectively, their evaluation may cause the invocation of method `overflow` if the promoted type of the operand is an arithmetical type.

Postfix Operators `++` and `--`

An unary expression followed by the postfix operator `++` (`--`) is called a postfix increment (decrement) expression. Note, that the prefix operators can only be applied to an expression that evaluates to a variable, otherwise a compile-time error occurs. The type of a prefix expression is the type of the operand and *not* the promoted type of the operand.

The intended semantics of a postfix increment (decrement) expression is to increment (decrement) the value of the operand variable by one. The value of a postfix increment (decrement) expression is the value *before* the increment (decrement). Especially, this is important if the increment (decrement) expression is part of a compound expression.

Evaluating a postfix increment (decrement) expression, the operand is increased (decreased) by one using the JAVA addition (subtraction) operator with the operand and the integer literal `1` as arguments. If necessary, the sum is narrowed by a primitive narrowing conversion to the type of the variable before it is stored.

Since postfix expressions are evaluated by internally performing a JAVA addition or subtraction, respectively, their evaluation may cause the invocation of method `overflow` if the promoted type of the operand is an arithmetical type.

Bitwise Operators `<<`, `>>`, `>>>`, `&`, `|`, `^`, and `~`

As mentioned in the introduction of this section, we do not allow bit operators to be applied to variables of the arithmetical type. The reason for this decision is, that the semantics of bit operators strongly depends on the used binary representation and of the number of bits used for storage. Since the arithmetical types have an infinite range and we do not define a binary representation for them, we think, it is reasonable to prohibit these operators for the arithmetical types.

For the semantics of the bit operators applied to the built-in primitve JAVA types, we refer to the JAVA language specification.

If the operands of the operators `&`, `|`, and `^` are of type `boolean`, then the operators are interpreted as logical operators (see next section).

Logical Operators `&`, `|`, and `^`

If the operands of the operators `&`, `|`, and `^` are of type `boolean`, then the type of the logical operator expression is `boolean`.

For the binary operator `&`, the result value is `true` if and only if both operands evaluate to `true`.

For the binary operator `|`, the result value is `true` if and only if at least one of the operands evaluates to `true`.

For the unary operator `^`, the result value is `true` if and only if the operand evaluates to `false`.

Conditional Operator `? :`

A conditional expression consists of three expressions. The first expression has to be of type `boolean`, otherwise a compile-time error occurs.

To determine the type of a conditional expression, the following rules apply in order.

1. If the second and third operands are of the same type, then this type is the type of the conditional expression.
2. Otherwise, if one of the second or third operand is of type `byte` and the other is of type `short`, then the type of the conditional expression is `short`.
3. Otherwise, if one of the second or third operand is of type `byte` or `arithByte` and the other is of type `arithShort`, then the type of the conditional expression is `arithShort`.
4. Otherwise, if one of the second or third operand is of type T where T is `byte`, `short`, `arithByte`, or `arithShort` and the other operand is a constant expression of type `int` or `arithInt` whose value is representable in T, then the type of the conditional expression is T.
5. Otherwise, the type of the conditional expression is the promoted type of the second and third operand.

At run time, the first operand is evaluated. If it evaluates to `true`, then only the second operand is evaluated and converted to the type of the conditional expression. Otherwise, if the first operand evaluates to `false`, only the third operand is evaluated an converted to the type of the conditional expression.

String Concatenation Operator `+`

It is not allowed to apply the string concatenation operator `+` to an operand of an arithmetical type because this operator is not an arithmetical operator.

For the semantics of the string concatenation operator applied to built-in JAVA types, we refer to the JAVA language specification.

Unary Logical Complement Operator `!`

The type of the operand must be `boolean`, otherwise a compile-time error occurs. The type of a logical complement expression is `boolean`.

The value of a logical complement expression is `true` if and only if the operand evaluates to `false`.

Binary Logical Conditional Operators `&&` and `||`

The type of the operands must be `boolean`, otherwise a compile-time error occurs. The type of a logical conditional expression is `boolean`.

Evaluating the operator `&&`, first, the left-hand operand is evaluated. If it evaluates to `false`, the right-hand operand is not evaluated and the result value of the expression is `false`. Otherwise, if the left-hand operator evaluates to `true`, the right-hand operand is evaluated and the result value is `true` if and only if the right-hand operand evaluates to `true`.

Evaluating the operator `||`, first, the left-hand operand is evaluated. If it evaluates to `true`, the right-hand operand is not evaluated and the result value of the expression is `true`. Otherwise, if the left-hand operator evaluates to `false`, the right-hand operand is evaluated and the result value is `true` if and only if the right-hand operand evaluates to `true`.

Simple and Compound Assignment Operators `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, and `>>>=`

For all of the assignment operators, the left-hand side must evaluate to a variable, otherwise a compile-time error occurs.

If the left-hand side operand is an array access, the semantics of the assignment operators is very involved since the evaluation of the array access may cause several exceptions. In the following we assume, that the left-hand side has already been evaluated without causing any exception and in thus can be treated as a variable. Since in this thesis we do not allow reference types, the right-hand side is always assignment compatible to the left-hand side (otherwise a compile-time error had occured) and thus, no `ArrayStoreException` may be thrown if the left-hand side is an array access.

- **Simple assignment operator `=`**

  If the evaluation of the right-hand operand terminates abruptly, the whole assignment expression terminates abruptly and no assignment occurs. Otherwise, the result value of the evaluation of the right-hand side is converted using an identitiy or widening primitive conversion to the type of the left-hand side and assigned to it.

– **Compound assignment operators `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, and `>>>=`**

The compound assignment operators `&=`, `^=`, `|=`, `<<=`, `>>=`, and `>>>=` are not allowed to be applied to operands of arithmetical types for the same reason as the bit operators `&`, `^`, `|`, `<<`, `>>`, and `>>>` are not allowed for arithmetical types.

If the left-hand side has no side-effects, the compound assignment operators are abbreviations and the expression `a` *op*`=` `b` is semantically equivalent to the expression `a = (`*T*`)(a` *op* `b)`, where *T* is the type of the expression `b`. If the left-hand side has side-effects, the above expressions are not equivalent because, using a compound assignment operator, the expression `a` is only evaluated once, whereas, using the simple assignment operator, `a` is evaluated twice.[3]

[3] For the exact semantics, if the left-hand side has side-effects, we refer to the JAVA language specification.

5 Different Semantics for Java Integer Arithmetic

In this chapter we motivate the extensions of syntax and the semantics presented in the previous chapter.

The generation of DL formulas from the UML/OCL specification and from the JAVA implementation poses the problem that two different semantics, especially concerning integer arithmetic, come into conflict. It took as a long time, until we arrived at the semantics $\mathcal{S}_{KeY}$ defined in the previous chapter and finally it turned out, that semantics $\mathcal{S}_{KeY}$ is a combination of two other semantics which we will present and discuss first.

Note, that the three semantics presented in the following are the DL semantics of programs and not the semantics implemented by the JAVA virtual machine and except for $\mathcal{S}_{Java}$, the DL semantics differs from the virtual machine semantics. The virtual machine exactly implements (or at least should) the semantics of JAVA as defined in the JAVA language specification.[1] This semantics is fixed and its fragment for integer arithmetic is called $\mathcal{S}_{JVM}$ in the following.

The first semantics for JAVA integer arithmetic we present is motivated by the semantics of the OCL and thus, it is called $\mathcal{S}_{OCL}$ in the following. In $\mathcal{S}_{OCL}$, each of the primitive integral JAVA types `byte`, `short`, `int`, and `long` is interpreted in the same way as the OCL type INTEGER. The arithmetical JAVA operators are interpreted in the same way as the arithmetical operators in the OCL (and thus as in mathematics). This means, in $\mathcal{S}_{OCL}$ the primitive JAVA types are interpreted as having an infinte range and overflow is totally disregarded. Thus, it is easy to see that the semantics $\mathcal{S}_{OCL}$ and $\mathcal{S}_{JVM}$ differ.

The difference between $\mathcal{S}_{OCL}$ and $\mathcal{S}_{JVM}$ becomes important when we are talking about the correctness of programs. Correctness means, that the program during its execution on a computer (in contrast to the symbolic execution in DL) meets or fulfills its specification. In the KeY project, the correctness of programs is formally proved by establishing the validity of DL formulas generated from the specification and implementation, which are called proof obligations. The programs occuring in DL formulas are interpreted using an operational semantics which ideally corresponds to the semantics defined in the JAVA language specification. But as semantics $\mathcal{S}_{OCL}$ shows, this does not apply if the wrong semantics is used and thus, the correct-

[1] Actually, the JAVA virtual machine interprets so called JAVA byte code which is generated by the JAVA compiler from the source code. Strictly speaking, it is the compiler that exactly implements the semantics defined in the JAVA language specification.

ness of a program must always be seen with respect to the used DL semantics of the implementation language. If a JAVA program is correct with respect to semantics $\mathcal{S}_{OCL}$, this does not guarantee that the program actually has the properties required in the specification during the execution on the JAVA virtual machine because the semantics $\mathcal{S}_{JVM}$ implemented by the virtual machine differs from semantics $\mathcal{S}_{OCL}$ used in the correctness proof. In this case, from the validity of the DL proof obligations, one cannot conclude that the program actually meets the specification and thus, semantics $\mathcal{S}_{OCL}$ is not acceptable for verifying software.

The second semantics, which we will call $\mathcal{S}_{Java}$, exactly corresponds to the semantics defined in the JAVA language specification [Bracha et al., 2000] and thus, is equivalent to the semantics $\mathcal{S}_{JVM}$ implemented by the virtual machine. Especially, this means that the primitive integral types `byte`, `short`, `int`, and `long` are interpreted as being finite and overflow is handled correctly. Semantics $\mathcal{S}_{Java}$ is for example used for the JAVA CARD DL presented in [Stenzel, 2001]. In contrast to semantics $\mathcal{S}_{OCL}$, using semantics $\mathcal{S}_{Java}$, from the validity of the DL proof obligations, one can conclude that all specified properties hold during the execution of the program on the virtual machine as well, since the DL semantics $\mathcal{S}_{Java}$ is equivalent to the virtual machine semantics $\mathcal{S}_{JVM}$. This means, using $\mathcal{S}_{Java}$, the notions "validity" of proof obligations (generated from the specification and the implementation) and "correctness" of the implementation coincide. Thus, one might think semantics $\mathcal{S}_{Java}$ is the best choice. But $\mathcal{S}_{Java}$ has also some drawbacks which will be discussed in the following.

To specify properties about integer numbers in the OCL, the type INTEGER is used. On the other hand, in the implementation usually the JAVA type `int`[2] is used for integer arithmetic. Thus, the data type `int` with finite range is used in the implementation of the program that is supposed to fulfill properties from the specification stated using the type INTEGER with infinte range. If one is not aware of that fact during the specification and implementation, this might lead to an implementation that is only correct "by accident".

By a program being correct only "by accident", we mean, that a program fulfills its specification although overflow may occur, but the fact that overflow occurs was not intended by the modeler and the programmer and thus, the behaviour of the program during the execution may differ from the intended behaviour. This situation mostly occurs, if the specification is incomplete or consists only of some safety constraints. Of course, it is better if a program is only "incidentally" correct than it is incorrect, but we think "incidentally" correct programs are also a source of error, as the following example shows. Again, we specify a method that is supposed to double the value of the argument. Suppose, instead of a complete specification of this method we only have the following specification that consists of some safety constraints that must be satisfied by an implementation.

[2] The other integral primitve types `byte`, `short`, and `long` could be used as well but as the name INTEGER suggests, mostly `int` is used.

```
context Math::double(arg:Integer):Integer
post: (even(arg@pre) implies even(result)) and
       arg@pre <> result
```

This specification states, that the result has to be even if the argument was and that the result has to be different from the argument. The predicate `even(x)` actually does not exist in the OCL and is a short form for `x % 2 = 0`. Of course, each implementation that doubles the argument satisfies the above constraint.

Now, we give a naive implementation of the method, that should double the argument. In our opinion, this implementation is only "incidentally" correct (with respect to the specification consisting only of the safety constraints) if the programmer was not aware of overflow.

```
public int double(int arg) {
  int res=arg*2;
  return res;
}
```

In many cases, the above program actually works in the supposed manner and satisfies the safety constraints and also returns the double of the argument. But, for example, running the method with an argument value of $\texttt{MAX\_int} - 1$, the execution of the statement `res=arg*2;` causes overflow. In this case, the return value is different from the argument and even as well. But the result is even only because overflows luckily preserves the property that the product of two even numbers is even as well.

A program that is only "incidentally" correct is always a possible source of error in the software development process and thus, this case should be avoided. Suppose, that the specification is extended and now requires, that the return value has to be positive if the argument value was. The programmer of the above implementation likely believes that his method fulfills the extended specification without any changes. But he will fail to prove the correctness because in the case that overflow occurs, the method returns a negative value although the argument was positive. In this case, it is very hard to find the reason why the implementation is not correct because the program does not work in the manner supposed by the programmer.

Of course, the problems mentioned above are not problems arising directly from the semantics $\mathcal{S}_{Java}$ itself but rather from the combination of the specification language UML/OCL and the implementation language Java. But since these languages are used in the KeY project, the problems arising from this combination have to be considered in our semantics for integer arithmetic.

A real disadvantage of $\mathcal{S}_{Java}$ itself is the fact that formulas, that are intuitively valid in mathematics like $\forall x \exists y (y > x)$ are not valid any more if x, y are of a built-in Java type like for example `int`. Also, intuitively valid DL formulas like $\forall x \{\mathtt{i} \leftarrow x\} \langle \mathtt{i=i;} \rangle \mathtt{i} \doteq \mathtt{i}$ are not valid if `i` is of a built-in type and x is of type $\mathbb{Z}$.

Now, that we have explained why, in our opinion, both semantics $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$ are not suitable for software verification, we present our approach. Of course, we want to maintain the property of $\mathcal{S}_{Java}$ that a verified program (this means, that the proof obligations are valid DL formulas) fulfills the specified properties during the execution. But we also want to prevent that programs are only "incidentally" correct (due to unintended overflow) and we want that intuitively valid formulas in mathematics like $\forall x \exists y (y > x)$ or intuitively valid DL formulas like $\forall x \{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i;}\rangle\mathtt{i} \doteq \mathtt{i}$ are valid in DL as well. To achieve that, we extend the syntax of JAVA by the additional primitve integral data types `arithByte`, `arithShort`, `arithInt`, and `arithLong` which are called arithmetical types in contrast to the built-in types `byte`, `short`, `int`, and `long`. The exact semantics for the additional types and the operators of JAVA integer arithmetic is defined in the previous chapter and is called semantics $\mathcal{S}_{KeY}$. The first obvious difference between semantics $\mathcal{S}_{KeY}$ on the one hand and semantics $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$ on the other hand, is that the signatures of the underlying programming languagues differ since $\mathcal{S}_{KeY}$ is a semantics for our extended JAVA, whereas $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$ are semantics for standard JAVA.

The additional arithmetical types basically have an infinite range. The operators acting on them have the same semantics as in $\mathcal{S}_{OCL}$ (and thus, as in mathematics) with the following restriction. If the values of the arguments of an operator are in valid range (this means, they are representable in the corresponding built-in types) but the result would not (this means, overflow occurs replacing the arithmetical types with the corresponding built-in types), then the result is calculated by an invocation of the unspecified method `overflow`, where the last parameter of the method invocation indicates the operator of the expression that caused the method invocation and the first parameters contain the argument values of the operands. This means, the result is unspecified and in particular, the execution of the method `overflow` does not have to terminate.

In the following, we explain, why we leave the method `overflow` unspecified in semantics $\mathcal{S}_{KeY}$.

The first reason is, that there is no reasonable implementation for the method `overflow`. In principle, there are three possible implemenations, that we shortly discuss in the following (a concrete implementation of `overflow` leads to a semantics that is an "instance" of $\mathcal{S}_{KeY}$):

- The implementation of the method `overflow` is such that it returns the value of the mathematical result of the operation and the execution of `overflow` always terminates. In this case, the same problems as in semantics $\mathcal{S}_{OCL}$ occur and thus, this implementation is not acceptable.

- The implementation of `overflow` calculates the result of the operation as if the operator had the same semantics as defined in the JAVA language specification and the execution of `overflow` always terminates. The instance of semantics $\mathcal{S}_{KeY}$, resulting from this implementation, is called $\mathcal{S}_{JVM'}$. As the name $\mathcal{S}_{JVM'}$

suggests, semantics $\mathcal{S}_{JVM'}$ is similar to $\mathcal{S}_{JVM}$ (the differences will be discussed later). The problem of $\mathcal{S}_{Java}$, that programs are only “incidentally” correct, also applies to $\mathcal{S}_{JVM'}$. The effects of “incidentally” correct programs have been shown in the discussion of $\mathcal{S}_{Java}$. An advantage of semantics $\mathcal{S}_{JVM'}$ compared to $\mathcal{S}_{Java}$ is that, using the arithmetical types, formulas like $\forall x \exists y (y > x)$ or $\forall x \{\mathtt{i} \leftarrow x\} \langle \mathtt{i=i;} \rangle \mathtt{i} \doteq \mathtt{i}$ are valid.

– The implementation of `overflow` is such that it never terminates. This implementation is not acceptable, because in this case, a formula like

$$[\mathtt{arithInt\ i=MAX\_int+1;}]\mathtt{i} \doteq 5$$

is (trivially) valid because the program in the box operator does not terminate due to overflow and the invocation of method `overflow`. But the execution of the program on the Java virtual machine terminates and thus, the same problem as in $\mathcal{S}_{OCL}$ occurs, namely that programs contained in valid DL formulas in general do not satisfy the specified properties during the execution on the virtual machine.

As one can see, each of the three possibilities leads to a semantics that we do not want to have, as explained previously. Thus, the first reason, why we leave `overflow` unspecified is, that there is no reasonable implementation for it. The second reason concerns the connection between the derivability of a formula in the calculus presented in Chapter 6 and the behaviour of a program contained in this formula during the execution on the Java virtual machine.

If a DLJ formula ϕ is derivable in our calculus based on $\mathcal{S}_{KeY}$ (this means, `overflow` is unspecified), then ϕ is valid in $\mathcal{S}_{KeY}$ for all implementations of `overflow`. This follows from the soundness of the calculus.

If ϕ is derivable and thus valid in semantics $\mathcal{S}_{KeY}$, then ϕ is valid for all implementations of `overflow`. In particular, ϕ is a valid formula for the second implementation of `overflow` mentioned above. Thus, ϕ is valid in semantics $\mathcal{S}_{JVM'}$ and, what is very important, one knows that no overflow occurs during the execution of p. Note, that the latter information is a conclusion from the validity of ϕ in semantics $\mathcal{S}_{KeY}$. If one only knows that ϕ is valid in semantics $\mathcal{S}_{JVM'}$, this conclusion is not correct. However, the information, that no overflow occurs, is very important and is used in Theorem 5.4.4.

But what one is really interested in, is whether a program p, contained in a formula ϕ, fulfils the specified properties during the execution on the Java virtual machine. As mentioned in the discussion of $\mathcal{S}_{Java}$, this applies to any program contained in a valid formula in $\mathcal{S}_{Java}$. To see, whether this also applies to a program contained in a formula valid in $\mathcal{S}_{KeY}$, we have to dwell on the differences between $\mathcal{S}_{JVM}$ and $\mathcal{S}_{JVM'}$.

Of course, there is a difference between $\mathcal{S}_{JVM}$ and $\mathcal{S}_{JVM'}$ concerning the signature of the underlying programming language. Semantics $\mathcal{S}_{JVM}$ is the semantic for the standard JAVA language, whereas $\mathcal{S}_{JVM'}$ (as $\mathcal{S}_{KeY}$) is a semantics for our extended JAVA. The additional types of the extended JAVA have an infinite range, and thus, in a DL based on $\mathcal{S}_{JVM'}$, there are states that do not exist in the JAVA virtual machine implementing $\mathcal{S}_{JVM}$. In the following, we call such states "unreal" states[3], whereas states, that exist in both DLs, are called "real" states. In DL based in $\mathcal{S}_{JVM'}$, a program can be enforced to reach an unreal state using updates.[4] For example, the program in the formula $\exists x(x > \texttt{MAX\_}T \wedge \{\texttt{i} \leftarrow x\}\langle\texttt{j=i;}\rangle\texttt{j} \doteq \texttt{i})$, which is valid in $\mathcal{S}_{KeY}$, starts in a unreal state where `i` has a value greater than $\texttt{MAX\_}T$. But we are interested in the behaviour of a program during the execution on a JAVA virtual machine, and thus, it does not matter, if a formula holds in an unreal state or not, since unreal states cannot be reached by the virtual machine. In real states, the semantics $\mathcal{S}_{JVM}$ and $\mathcal{S}_{JVM'}$ are equivalent (in both, the result of overflow is the same). In particular, this means, that in real states the semantics $\mathcal{S}_{JVM}$ of the built-in types corresponds to the semantics $\mathcal{S}_{JVM'}$ of the arithmetical types and thus, in real states a program p is equivalent to a program $ptransf(p)$, where the arithmetical types are replaced with the corresponding built-in types (see Theorem 5.4.5).

Corollary 5.4.6 summarises the considerations of the previous paragraphs. It states that, if the formula $\Gamma \to \langle p \rangle \psi$ derivable in our calculus and program p is started in a real state s satisfying $s \models_{\mathcal{S}_{KeY}} \Gamma$, no overflow occurs during the execution of the transformed program $ptransf(p)$ on the JAVA virtual machine and after the execution, the property ψ holds. This case is depicted in Figure 5.1, where the dashed arrows symbolise the state transitions of the virtual machine and the solid arrows symbolise the state transitions in DL based on $\mathcal{S}_{JVM'}$. As one can see, all states reached are real states, symbolised by the dark circle (the white circle are unreal states).

Note, that Corollary 5.4.6 does not apply to arbitrary formulas because a formula like $[p]true$ is always derivable, no matter whether overflow occurs during the execution of p or not. However, the generation of proof obligations from the specification and implementation typically results in formulas of the form $\Gamma \to \langle p \rangle \psi$.

As mentioned above, a disadvantage of semantics $\mathcal{S}_{Java}$ is that formulas, that are intuitively valid over the integer numbers $\mathbb{Z}$ like $\forall x \exists y(y > x)$, are not valid if the variables are of an integral JAVA type, like `int`, because of their finiteness. This disadvantage does not apply to the arithmetical types since their range is infinite, and the above formula is valid if the variables are of an arithmetical type.

In $\mathcal{S}_{KeY}$, the semantics of the built-in types and the operators acting on them exactly corresponds to semantics $\mathcal{S}_{Java}$ and thus to the definitions in the JAVA language

[3] The name "unreal" suggests that those states are not reachable by the execution of a program on the virtual machine. Intuitively, in "unreal" states, the value of at least one variable of an arithmetical type is not representable in a variable of the corresponding built-in type. For example, if a variable of type `arithByte` in a state has a value of 200, then this state is said to be unreal because the maximum value of variables of the corresponding type `byte` is 127.

[4] Another possibility that an unreal state may be reached, is when the method `overflow` is invoked due to overflow.

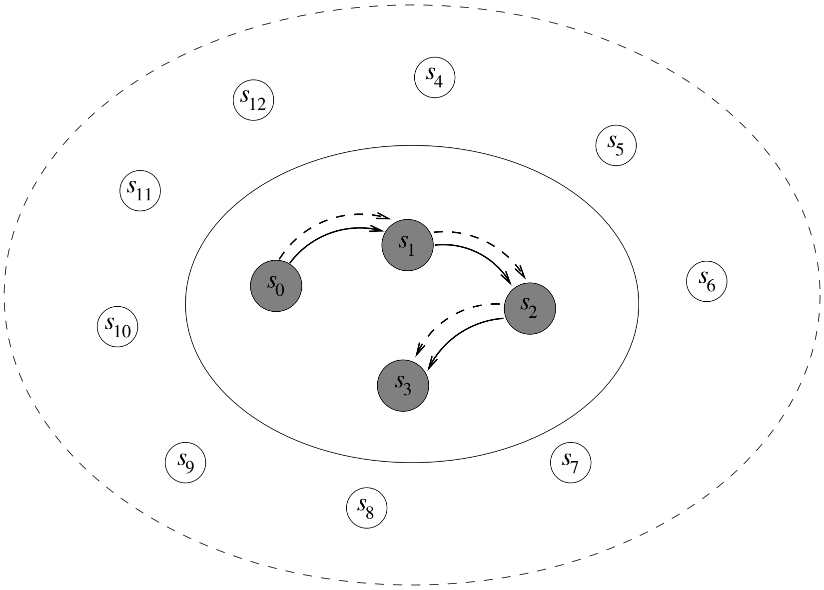


Figure 5.1: The starting state in DLJ is a real state.

specification. Thus, using the built-in types, it is still possible to make use of the effects of overflow. On the other hand, using the arithmetical types, the danger of unintended overflow is eliminated.

Above, we mentioned, that the method `overflow` is invoked if the result value of an operation of an arithmetical type is not representable in the corresponding built-in type (if a value of a variable of an arithmetical type is not representable in the corresponding built-in type, in the following we say, that the variable is not *in valid range*). But there is an exception, namely if at least one argument of the operation is not in valid range. In this case, method `overflow` is not invoked. The reason for that is, that a state, in which this situation occurs, is always an unreal state and thus, this does not affect the actual execution of a program because such a state cannot be reached by the virtual machine. For example, the formula $\exists x(x > \mathtt{MAX\_}T \wedge \{\mathtt{i} \leftarrow x\}\langle \mathtt{j=i+1;}\rangle \mathtt{j} \doteq \mathtt{i}+1)$ is valid in $\mathcal{S}_{KeY}$ if x and `i,j` are of an arithmetical type (for example `arithInt`). In this case, method `overflow` is not invoked during the symbolic execution in DL, since the state, in which the addition is evaluated, is an unreal state. Therefore, we cannot make any proposition about the behaviour of the program after replacing the arithmetical types with the corresponding built-in types. In fact, in this case trying to make a proposition about the behaviour of the transformed program during the execution of the Java virtual machine does not make any sense because the situation, that a unreal state is reached, cannot occur during the execution on the virtual machine.

In Table 5.1, the properties of the three semantics $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, and $\mathcal{S}_{KeY}$ are sum-

Property	$\mathcal{S}_{OCL}$	$\mathcal{S}_{Java}$	$\mathcal{S}_{KeY}$
Underlying programming language	Java	Java	extended Java
Overflow on built-in integer types	no	yes	yes
Overflow on arithmetical types	—	—	in unreal states
Range of built-in integer types	infinite	finite	finite
Range of arithmetical types	—	—	infinite
Existence of unreal states	yes	no	yes
Behaviour of programs in DL and on the JVM	different	equal	equal under certain conditions

Table 5.1: Comparison of some properties of $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, and $\mathcal{S}_{KeY}$.

Formula	$\mathcal{S}_{OCL}$	$\mathcal{S}_{Java}$	$\mathcal{S}_{KeY}$
$\forall x \exists y (y > x)$	`int`	—	`arithInt`
$\exists x (x > 0 \rightarrow \{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i+1;}\rangle \mathtt{i} < 0)$	—	`int`	`int`
$\forall x \{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i+1;}\rangle \mathtt{i} \doteq \mathtt{i} + 1$	`int`	—	—
$\forall x \{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i+1-1;}\rangle \mathtt{i} \doteq \mathtt{i}$	`int`	`int`	`int`
$\forall x \{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i;}\rangle \mathtt{i} \doteq \mathtt{i}$	`int`	—	`arithInt`
$\forall x (even(x) \rightarrow \{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i+2;}\rangle even(\mathtt{i}))$	`int`	`int`	`int`

Table 5.2: This table shows some formulas and in the last three columns is denoted of which type the program variable `i` has to be that the according formula is valid in semantics $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, or $\mathcal{S}_{KeY}$. The logical variables x and y are of type $\mathbb{Z}$.

marised and compared.

Some sample formulas and the necessary type to make the according formula valid in the different semantics are depicted in Table 5.2.

Before, in the following sections, we present the three mentioned semantics in detail, we summarise the main features of semantics $\mathcal{S}_{KeY}$.

– It is still possible to make use of the effects of overflow by explicitly using the primitive built-in Java types in the specification as well as in the implementation.

– Unintended overflow is a common source of error which, following our approach, can by avoided by using the additional arithmetical types.

– Formulas like $\forall x \exists y (y > x)$ or $\forall x \{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i;}\rangle \mathtt{i} \doteq \mathtt{i}$ that are intuitively valid over the integer numbers $\mathbb{Z}$, are still valid if x, y and `i` are of an arithmetical type.

To conclude the motivation of our semantics $\mathcal{S}_{KeY}$, we shortly explain the steps in software development following our approach.

1. *Specification:* In the UML/OCL specification, the type OCL type INTEGER is used.

2. *Implementation:* If an operation is specified using INTEGER, in the implementation, the arithmetical types `arithByte`, `arithShort`, `arithInt`, or `arithLong` are used.

3. *Verification:* Using the calculus presented in [Beckert, 2001] extended by the rules in Chapter 6, one has to derive the proof obligations generated from the specification and implementation using the translation described in [Keller, 2002]. If all proof obligations are derivable, from Corollary 5.4.6 follows, that the program, if the requirements of the corollary are satisfied, after replacing the arithmetical types with the corresponding built-in types, satisfies all specified properties during the execution on the JAVA virtual machine and in particular, no overflow occurs.

5.1 Definitions

To be able to compare and describe the three different semantics and their consequences concerning correctness, we first have to define some notions.

In the previous section, we mentioned *real states* and *unreal states*. In the following, these notions are needed to compare the symbolic execution of a program in DL using one of the semantics $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$ and $\mathcal{S}_{KeY}$ to the execution of the program on the JAVA virtual machine which implements the semantics $\mathcal{S}_{JVM}$. Unreal states are states that exist in DL but which are not reachable by the virtual machine. For example, the virtual machine can never reach a state in which a variable of the built-in type `byte` has a value greater than 127. Since semantics $\mathcal{S}_{Java}$ exactly corresponds to the semantics defined in the language specification and thus to the semantics implemented by the virtual machine, in a DL based on $\mathcal{S}_{Java}$ there are no unreal states. To define real and unreal states, first we need the notion *valid range*.

Definition 5.1.1 (Valid range)
The value val of a variable or attribute of a built-in JAVA *type* T *is in* valid range *if and only if val satisfies the inequations val* $\geq$ `MIN_`T *and val* $\leq$ `MAX_`T.

The value of a variable or attribute of an arithmetical type is in valid range if and only if the value is in valid range of the corresponding built-in type.

Definition 5.1.2 (Real states)
Let s be an arbitrary DLJ state.

State s is called a real state *if all variables and attributes of an arithmetical type, s depends on, are in valid range.*

Definition 5.1.3 (Unreal states)
A state, that is not a real state, is called an unreal state*.*

In order to illustrate the differences between the execution of a program in a DL based on the semantics $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, and $\mathcal{S}_{KeY}$ in contrast to the execution on the JAVA virtual machine, we will use figures similar to Figure 5.2.

By means of Figure 5.2 we want to explain the meaning of the circles and arrows. The white circles in the figures labelled with $s_4, s_5, \ldots, s_{12}$ symbolise a subset of the set of unreal states and the dark circles s_0, s_1, s_2, s_3 are elements of the set of real states. To illustrate which states are reached during the execution of a program p in DL and on the virtual machine, DL state transitions are depicted with dashed arrows and for state transitions of the virtual machine we use solid arrows. By the execution of a program in DL, we mean the symbolic execution of the program which results in state transitions depending on the used semantics for the programming language JAVA.

5.2 Semantics $\mathcal{S}_{OCL}$

In this section we present a semantics for JAVA integer arithmetic which is used in many publications on the formalisation of JAVA semantics, for example in [Huisman, 2001]. This semantics, which we call $\mathcal{S}_{OCL}$, is the most simple semantics of the three given here. Especially, this means that problems concerning overflow are totally ignored.

In $\mathcal{S}_{OCL}$, the arithmetical operators of JAVA have the same semantics as in mathematics and therefore as in OCL. The primitive JAVA types `byte`, `short`, `int`, and `long` are all interpreted as having an infinite range and thus correspond to the OCL type INTEGER.

Semantics $\mathcal{S}_{OCL}$ is chosen very often because it is simple and if one does not want to be concerned with details like overflow and finiteness of data types this semantics is the first choice.

An advantage of $\mathcal{S}_{OCL}$ is, that it is easy to give a calculus for integer arithmetic because any arithmetical JAVA expression, free of side-effects, can simply be translated into a logical term since the arithmetical JAVA operators have the same semantics as the corresponding logical functions.

In the following, by relevant states we mean the states in which a specified property must hold. In case of a OCL postcondition, the (single) relevant state is the final state of the execution of a program. In case of an OCL invariant, the relevant states are all states reached during the execution of the program.[5]

[5] Since the DL modalities "box" and "diamond" only refer to the final states of a program execution, classical DL is not suitable for verifying invariants or safety constrains. In [Beckert and Schlager, 2001], an extension of DL with so called trace modalities "throughout" and "at least once", which refer to *all* states reached during the execution, is defined.

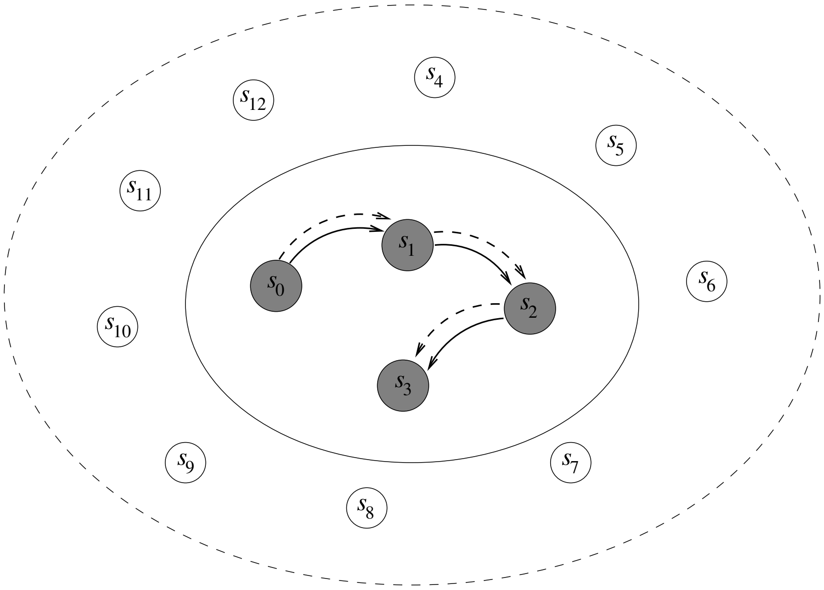


Figure 5.2: All states reached are real states.

Now we will distinguish several cases which may occur when comparing the states reached during the execution of a program p on the JAVA virtual machine and the states reached during the symbolic execution of p in DL based on $\mathcal{S}_{OCL}$. We will also dwell on the impact of $\mathcal{S}_{OCL}$ on the correctness of programs. Especially, this is interesting if the sequence of states reached in DL differs from the sequence of states reached on the JAVA virtual machine. In this case, from the validity of the DL proof obligations does not follow that the program fulfils the specified properties during the execution on the virtual machine.

- All states reached during the execution of the program p are real states (see Figure 5.2).

  Since all states reached during the symbolic execution of p in DL are real states, the execution of p on the JAVA virtual machine exactly reaches the same states. Especially, this means that no overflow occurs because otherwise, in DL at least one unreal state would be reached. An example for this situation is the program `int i=0; i=i+1;`.

  In this case, if the DL proof obligation are valid formulas, then the specified properties actually hold during the execution on the virtual machine.

- During the execution in DL unreal states are reached. The set of reached unreal states may also contain the starting and the final state (see Figure 5.3).

In Figure 5.3, state s_1 is a real state but the next state s_5 reached by the execution in DL is an unreal state. This means that the statement executed in state s_1 causes overflow. In this case, the sequence of states reached during the execution in DL differs from the sequence of states reached by the execution on the virtual machine.

Discussing the impact on correctness, we have to distinguish several cases. Suppose, ϕ is a property from the specification that is supposed to hold during (if ϕ is an invariant) or after (if ϕ is a postcondition) the execution of a program p. To all of the next four distinguished cases applies, that from the validity of the DL proof obligations in general does not follow that the program p satisfies the property ϕ during the execution on the JAVA virtual machine since unreal states are reached in DL.

- During the execution in DL, ϕ holds in all reached relevant unreal states (either in all states if ϕ is an invariant or in the final state in case of a postcondition) and ϕ holds in all relevant (real) states during the execution on the virtual machine.

  In this case, the program is judged being correct (because the according DL formula is valid) and in fact, the program satisfies the specified property ϕ during the execution on the virtual machine.
- During the execution in DL, ϕ does not hold in all reached relevant unreal states but ϕ holds in all relevant (real) states during the execution on the virtual machine.

  In this case, the validity of the according DL formula cannot be established and thus, the program is said to be not correct. However, running the program on the virtual machine, the property ϕ holds. Thus, the program is said to be incorrect spuriously.
- During the execution in DL, ϕ holds in all reached relevant unreal states but ϕ does not hold in all relevant (real) states during the execution on the virtual machine.

  Since the validity of the according DL formula can be established, the program is spuriously said being correct because the property ϕ does not hold during the execution on the virtual machine. Thus, this is the worst case that may happen.
- During the execution in DL, ϕ does not hold in all reached relevant unreal states and ϕ does not hold in all relevant (real) states during the execution on the virtual machine.

  In this case, the program cannot be verified and in fact, the property ϕ does not hold during the execution on the virtual machine.

• A run of the program p on the virtual machine does not terminate but the symbolic execution in DL does (see Figure 5.4). The non-termination is sym-

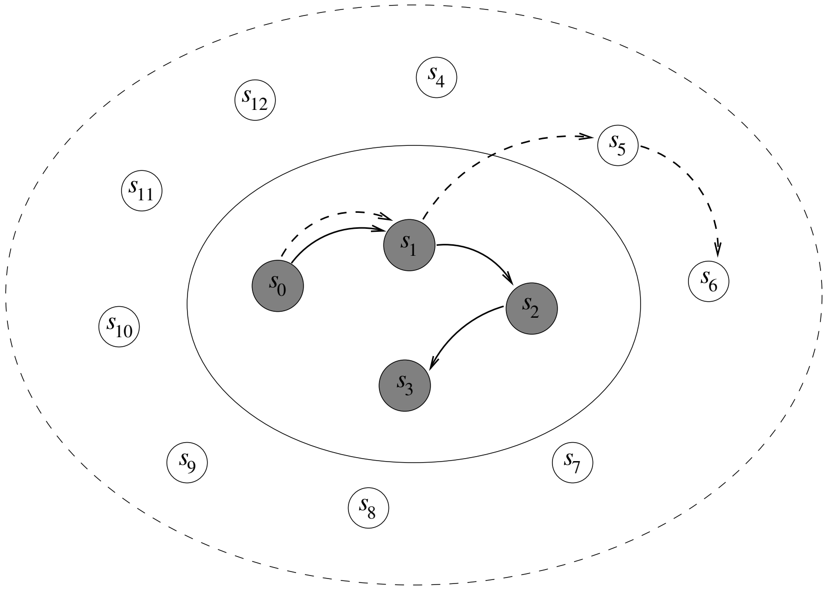


Figure 5.3: During the execution of the program p on the virtual machine overflow occurs and thus, in DL unreal states are reached.

bolised by the bold arrows which suggest an endless loop that always traverses the same states.

An example for this situation is the following program.

```
for (byte b=-128; b<=127; b++)
  System.out.println(b);
```

This for-loop does not terminate on the virtual machine. If the variable `b` has the value 127 the loop condition still holds and thus, the increment operator is applied to `b` and the loop body is executed. But the increment of `b` causes overflow and thus, `b` has the value -128 and the loop starts again.

If the specified properties hold in the relevant states reached during the symbolic execution in DL, the program is said to be totally correct although a run of the program on the virtual machine does not even terminate.

- A run of the program on the virtual machine terminates but the symbolic execution in DL does not (see Figure 5.5).

  The following program is an example for this situation.

```
int i=1;
while (i>0)
  i++;
```

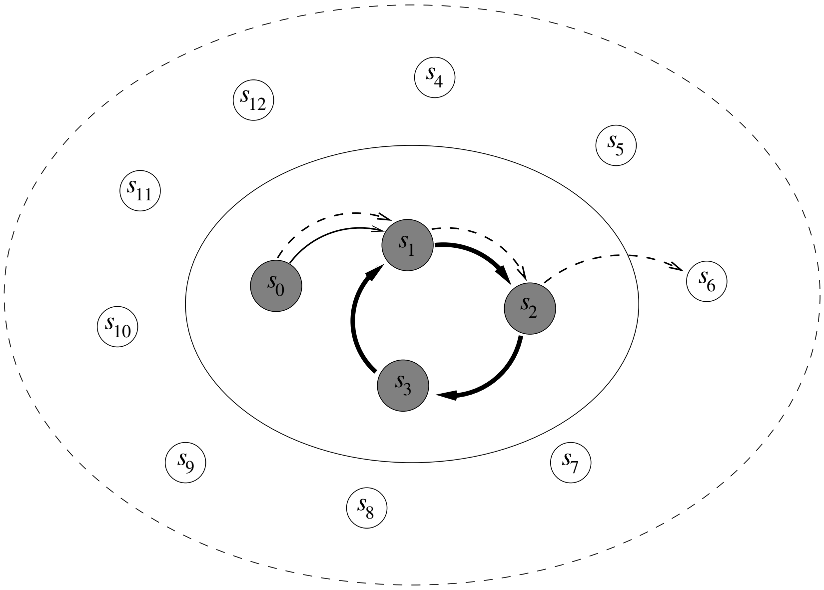


Figure 5.4: The execution of the program on the virtual machine does not terminate but the execution in DL does.

If this program is executed on the JAVA virtual machine it terminates because sometime overflow occurs and the loop condition does not hold any more. However, the execution in DL does not terminate since no overflow occurs and the loop condition always holds.

5.2.1 Discussion of $\mathcal{S}_{OCL}$

The investigation of the possible cases that may occur using semantics $\mathcal{S}_{OCL}$ in the previous section shows that this semantics is not suitable for the verification of software.

The main drawback of $\mathcal{S}_{OCL}$ is, that programs can be verified in DL (this means the validity of the according DL proof obligations can be established) but during the execution on the JAVA virtual machine, the specified properties do not hold in general. As we have seen in the case depicted in Figure 5.4, programs may be judged being totally correct which in fact do not even terminate. A minor drawback is that some programs cannot be verified although they fulfil the specified properties during the execution on the virtual machine.

As a final example of the impact of semantics $\mathcal{S}_{OCL}$ on correctness, we want to take a look at our introducing specification example in Section 3.1 and the possible implementation in Section 3.2. The postcondition

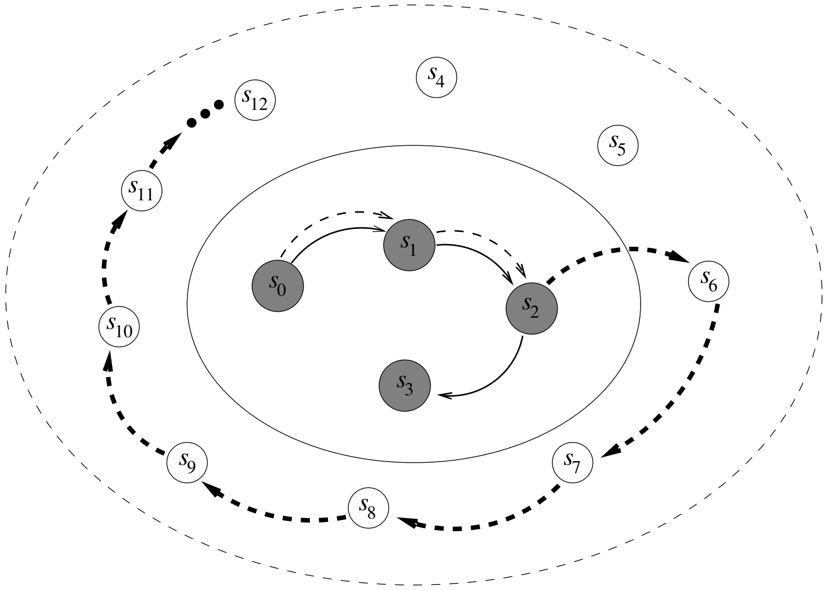


Figure 5.5: The execution of the program on the virtual machine terminates but the execution in DL does not.

```
context Math::square(arg:Integer):Integer
post:  result=arg*arg
```

requires that the method `square` always returns the mathematical square of the argument `arg`. Without giving the proof, it is easy to see that the implementation given in Figure 3.2 is said to be correct in semantics $\mathcal{S}_{OCL}$. However, it is not difficult to find a value for the parameter `arg` for which overflow occurs in the body of the method and thus, the result of the method does not coincide with the mathematical square of the argument.

5.3 Semantics $\mathcal{S}_{Java}$

The second semantics $\mathcal{S}_{Java}$ for Java integer arithmetic we present, exactly corresponds to the semantics given in the Java language specification [Bracha et al., 2000]. Especially, this means that all effects of overflow are handled correctly as defined in the language specification.

Similar to $\mathcal{S}_{OCL}$, we also compare the sequence of states reached during the execution on the Java virtual machine and during the execution in DL. But in contrast to $\mathcal{S}_{OCL}$, here we have only one possible case since the semantics $\mathcal{S}_{Java}$, the DL is based on, exactly corresponds to the semantics $\mathcal{S}_{JVM}$ implemented by the virtual machine. This means, during the symbolic execution in DL no unreal states are reachable (they do not even exist in DL based on $\mathcal{S}_{Java}$). This case is depicted in 5.2.

Discussion of $\mathcal{S}_{Java}$

Semantics $\mathcal{S}_{Java}$ is exactly the same as the JAVA semantics defined in the language specification. Thus, one might think that this is the "right" semantics. And in fact in cannot happen, that one can verify a program that does not satisfy the specified properties during the execution on the JAVA virtual machine, as it is the case with $\mathcal{S}_{OCL}$.

But there are problems arising from the different interpretations of the OCL type INTEGER and the integral JAVA types like `int`. We believe that many modellers and/or programmers do not think about the finite range of the integral JAVA types and the consequences like overflow. As we have explained in the introduction of this chapter, this might result in programs that contain unintended overflow and even if they fulfil the specification, this is always a source of error in the ongoing software development process.

Another disadvantage of semantics $\mathcal{S}_{Java}$ is that formulas, that are intuitively valid over the integer numbers $\mathbb{Z}$, are not valid over the built-in primitive integral types like `int`. This applies as well to pure first-order formulas as to DL formulas containing simple programs. As an example, look at the formula $\forall x \exists y \{\mathtt{i} \leftarrow x\}\langle \mathtt{i=i;}\rangle y > \mathtt{i}$ which states that for every number x there exists a greater number y. The simple program in the formula does not change the value of `i` and could actually be simplified to the empty program (because the statement has no side-effects) without changing the semantics of the formula. Then, the empty diamond operator could be removed, resulting in the pure first-order formula $\forall x \exists y (y > x)$ which of course is valid if x, y are of type $\mathbb{Z}$. However, if x, y are of a built-in primitive JAVA type (for example `int`), this formula is not valid anymore because if x is instantiated with `MAX_int` there is no greater element of type `int`. We think, if such intuitively valid formulas cannot be verified, this might be confusing.

5.4 Semantics $\mathcal{S}_{KeY}$

The semantics $\mathcal{S}_{KeY}$ which we present in this section is the actual semantics used for integer arithmetic in the KeY project. Semantics $\mathcal{S}_{KeY}$ is based on an extended syntax of JAVA and can be seen as a mixture of $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$ trying to combine the advantages and eliminating the drawbacks of both semantics.

The exact definition of semantics $\mathcal{S}_{KeY}$ is given in Chapter 4. In the introduction of this chapter, $\mathcal{S}_{KeY}$ has already been motivated and discussed in detail. In the following, we present some theorems concerned with the connection between a program containing arithmetical types, verified using our calculus presented in Chapter 6, and the actual execution of the transformed program on the virtual machine.

One goal of our approach was to contribute to avoid errors arising from unintended overflow. To attain this goal, we extend the syntax of JAVA with the additional types `arithByte`, `arithShort`, `arithInt`, and `arithLong`. These types are called *arithmetical types*. If the arithmetical types are used in the implementation instead of the built-in JAVA types, during the correctness proof one has to show that no overflow[6] occurs.

Of course, the arithmetical types are not allowed to occur in a program that is supposed to be compiled and run on the virtual machine. Thus, the arithmetical types have to be replaced with the corresponding built-in types before the program can be compiled and executed.

Definition 5.4.1 (Arithmetical type transformation)
Let p be program that may contain arithmetical types.

The program ptransf(p) is the program p after replacing all arithmetical types with the according primitive JAVA *types.*

A consequence of applying the type transformation to a program is of course, that the program that is actually executed on the virtual machine differs from the program which was verified. The connection between the verified program and the translated program is stated by the following theorems.

Theorem 5.4.2 concerns typing and states that the program *ptransf*(p) is well-typed if the original program p was.

Theorem 5.4.2
If a JAVA *program p that may contain arithmetical types is well-typed, then the program ptransf(p), resulting from the arithmetical type transformation, is well-typed as well.*

Proof:
Reference types are not affected by the arithmetical type transformation. Thus, we only have to prove that expressions containing primitive types are well-typed.

According to the JAVA language specification, an assignment is well-typed, concerning the primitive types, if the right-hand side of the assignment is assignment compatible to the left-hand side. This is decidable at compile-time and a compile-time error occurs if the assignment compatibility is violated.

The type of the expression on the right-hand side of an assignment is determined by unary or binary promotion (see Section 4.2.3 or Sections 5.6.1 and 5.6.2 in the JAVA language specification [Bracha et al., 2000]).

[6] Actually, talking about the arithmetical types, "overflow" has a different meaning than usual because the arithmetical types have an infinite range. Overflow in this context means that the result of an operation is not in valid range and thus, replacing the arithmetical type with its corresponding built-in JAVA type, overflow would occur.

Let S be the promoted type of the right-hand side of the assignment and let T be the type of the left-hand side. If S is an arithmetical type then S' is the corresponding built-in JAVA type, otherwise S' is equal to S. Analogously, T' is the corresponding built-in type of T.

Now, we assume that S is assignment compatible to T. This means, according to Definition 2.2.4 that either $S = T$ or $S \overset{*}{\prec} T$.

Then, in Figure 2.3 it is easy to see that either $S' = T'$ or $S' \overset{*}{\prec} T'$ and thus, S' is assignment compatible to T'. ■

To describe the behaviour of a transformed program *ptransf*(p) during the execution on the JAVA virtual machine, one needs the notion *isomorphic state.*

Definition 5.4.3 (Isomorphic state)
Let s be a real DLJ state and let Var be the set of all program variables of an arithmetical type the state s depends on.

The isomorphic state $iso(s)$ *to s is the* JAVA *virtual machine state, in which all the variables in Var are of the corresponding built-in types and are assigned the same values as in s.*

If s is a real state, the existence of $iso(s)$ is guaranteed, since by definition, in real states the values of all variables of the arithmetical types are representable in the corresponding built-in types.

Theorem 5.4.4
Let p be a JAVA *program that may contain arithmetical types, let ptransf(p) be the transformed program, and let Γ and ψ be pure first-order predicate logical formulas.*

If $\vdash_{\mathcal{C}_{DLJ}} \phi$, then the following propositions hold:

1. $\models_{\mathcal{S}_{OCL}} \phi$.
2. $\models_{\mathcal{S}_{JVM'}} \phi$.

Proof:

Due to the soundness of the rules of our calculus, from $\vdash_{\mathcal{C}_{DLJ}} \phi$ follows $\models_{KeY} \phi$ for all possible implementations of method `overflow`.

In particular, $\models_{KeY} \phi$ holds for the concrete implementation of `overflow`, which terminates and whose return value is the same as the mathematical result of the operation causing overflow (this means $\models_{KeY} \phi$ holds for the instance $\mathcal{S}_{OCL}$ of $\mathcal{S}_{KeY}$). Thus, $\models_{\mathcal{S}_{OCL}} \phi$ holds.

The same applies to the implementation of `overflow` which terminates and returns the same result as the result of the operation causing overflow would be in $\mathcal{S}_{JVM}$ (this means $\models_{KeY} \phi$ holds for the instance of $\mathcal{S}_{JVM'}$ of $\mathcal{S}_{KeY}$). Thus, $\models_{\mathcal{S}_{JVM'}} \phi$ holds. ■

Theorem 5.4.5
Let p be a JAVA *program that may contain arithmetical types.*

Then, for all real states s and all (arbitrary) states s' holds:

$$\textit{If} \quad s\,[\![p]\!]_{\mathcal{S}_{JVM'}}\,s' \quad \textit{then} \quad iso(s)\,[\![ptransf(p)]\!]_{\mathcal{S}_{JVM}}\,iso(s').$$

Proof:
The existence of the isomorphic state $iso(s)$ to s is guaranteed by definition.

Semantics $\mathcal{S}_{JVM'}$ is defined such that each state reachable by an arbitrary program, starlet in a real state, is a real state as well. From that follows that state s' is a real state. In particular, semantics $\mathcal{S}_{JVM'}$ is defined such that in real states the semantics of the operators acting on the arithmetical types is equivalent to the semantics of the operators acting on the built-in types in $\mathcal{S}_{JVM}$. Since the programs p and $ptransf(p)$ only differ in the fact, that the arithmetical types in p are replaced by the corresponding built-in types in $ptransf(p)$ and the semantics of the arithmetical types coincides with the semantics of the built-in types in real states, it is obvious, that the state $iso(s')$, reached by the execution of the program $ptransf(p)$, using semantics $\mathcal{S}_{JVM}$, is the isomorphic state to s'. ■

The following corollary states, that, if $\Gamma \to \langle p\rangle\psi$ is derivable in our calculus and program p is started in a real state s with $s \models_{\mathcal{S}_{KeY}} \Gamma$, no overflow occurs during the execution of $ptransf(p)$ started in $iso(s)$. The meaning of $s \models_{\mathcal{S}_{KeY}} \Gamma$ is that Γ holds in s for all possible implementations of `overflow`.

Corollary 5.4.6
Let Γ, ψ be pure first-order predicate logical formulas, let p be an arbitrary JAVA *program that may contain arithmetical types, and let s be an arbitrary DLJ state.*

If

- $\vdash_{\mathcal{C}_{DLJ}} \Gamma \to \langle p\rangle\psi$,
- $s \models_{\mathcal{S}_{KeY}} \Gamma$,
- *s is a real state, and*
- *p is started in s*

then for all states s' with $s\,[\![p]\!]_{\mathcal{S}_{KeY}}\,s'$ it holds

$$s'_{\mathcal{S}_{KeY}} \models \psi.$$

Furthermore it holds, that no overflow occurs during the execution of the transformed program $ptransf(p)$ started in $iso(s)$ on the JAVA *virtual machine and $ptransf(p)$ terminates in state $iso(s')$.*

Proof:
First we prove the proposition $s'_{\mathcal{S}_{KeY}} \models \psi$.

Due to the soundness of the rules of the calculus, from $\vdash_{\mathcal{C}_{DLJ}} \Gamma \rightarrow \langle p \rangle \psi$ follows for all possible implementations of `overflow`

$$\models_{\mathcal{S}_{KeY}} \Gamma \rightarrow \langle p \rangle \psi. \tag{5.1}$$

From $s \models_{\mathcal{S}_{KeY}} \Gamma$ and (5.1) follows for all states s' with $s\,[\![p]\!]_{\mathcal{S}_{KeY}}\,s'$[7]

$$s' \models_{\mathcal{S}_{KeY}} \psi. \tag{5.2}$$

From the requirement that s is a real state and p is started in s follows from Theorem 5.4.5 that the transformed program $ptransf(p)$ started in state $iso(s)$ terminates in state $iso(s')$ on the Java virtual machine.

Finally, we prove the proposition that no overflow occurs during the execution of $ptransf(p)$ on the Java virtual machine.

We assume $\vdash_{\mathcal{C}_{DLJ}} \Gamma \rightarrow \langle p \rangle \psi$ but overflow occurs during the execution of p started in state s.

From (5.1) follows in particular for all possible implementations of `overflow`

$$s \models_{\mathcal{S}_{KeY}} \Gamma \rightarrow \langle p \rangle \psi. \tag{5.3}$$

From (5.3) and the requirement $s \models_{\mathcal{S}_{KeY}} \Gamma$ follows

$$s \models_{\mathcal{S}_{KeY}} \langle p \rangle \psi. \tag{5.4}$$

From the assumption that overflow occurs during the execution of p started in state s follows that there is an implementation of `overflow` with

$$s \not\models_{\mathcal{S}_{KeY}} \langle p \rangle \psi. \tag{5.5}$$

For example, such an implementation of `overflow` is

$$\texttt{while (true) \{ \};}. \tag{5.6}$$

If overflow occurs in p, then `overflow` is invoked which does not terminate and thus (5.5) holds.

(5.5) is a contradiction to (5.4) and thus, no overflow occurs during the execution of p in $\mathcal{S}_{KeY}$. From that follows that no overflow occurs during the execution of p in $\mathcal{S}_{JVM'}$ which is an instance of $\mathcal{S}_{KeY}$.

Finally, from Theorem 5.4.5 follows that no overflow occurs during the execution of $ptransf(p)$ on the Java virtual machine. ■

[7] $s\,[\![p]\!]_{\mathcal{S}_{KeY}}\,s'$ means, that there exists an implementation of `overflow` with $s\,[\![p]\!]_{\texttt{overflow}}\,s'$

5.4.1 Variants of Semantics $\mathcal{S}_{KeY}$

In semantics $\mathcal{S}_{KeY}$, we defined, that the method `overflow` does not have a known implementation and thus, `overflow` is unspecified. By giving axioms, that the method `overflow` must satisfy, it is possible to define variants of $\mathcal{S}_{KeY}$.

For example, one could define, that `overflow` always terminates or that `overflow` is symmetric. If `overflow` always terminates, a formula like $\Gamma \rightarrow \langle p\rangle \mathit{true}$ is possibly derivable, even if overflow occurs during the execution of p, since goals of the form

$$\Gamma \vdash \langle \texttt{x=overflow(arg1,arg2,op);}\rangle \mathit{true},\ \Delta$$

(resulting from the application of Rule 6.4.2) could immediately be closed with the information that the invocation of `overflow` terminates.

As long as the axioms are such that they are satisfiable by the instances $\mathcal{S}_{OCL}$ and $\mathcal{S}_{JVM'}$ of $\mathcal{S}_{KeY}$, Theorem 5.4.4 and Theorem 5.4.5 still hold.

Common to all variants is, that formulas containing programs, which might cause overflow in expressions of an arithmetical type, are possibly derivable. The consequence is, that Corollary 5.4.6 does not hold anymore because in this case, from the derivability of the formula $\Gamma \rightarrow \langle p\rangle\psi$ one cannot conclude that p does not cause overflow. For example, if the axiom is given, that `overflow` always terminates, the formula $\{\texttt{i} \leftarrow \texttt{MAX\_int}\}\langle\texttt{i=i+1-1;}\rangle\texttt{i} \doteq \texttt{MAX\_int}$ could be derived although overflow occurs in the program if the calculus contains the following rule.

$$\frac{\begin{array}{l}1.\ \Gamma \vdash \mathit{isdef}(\mathit{slhs}),\ \Delta\\ 2.\ \Gamma \vdash \mathit{isdef}(\mathit{srhs}),\ \Delta\\ 3.\ \Gamma \vdash \langle\pi\ \mathit{slhs}\texttt{=}\mathit{srhs}\texttt{;}\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \langle\pi\ \mathit{slhs}\texttt{=}\mathit{srhs}\texttt{+1-1;}\ \omega\rangle\phi,\ \Delta}$$

Note, that this rule is only correct if the axiom is given, that `overflow` always terminates. In $\mathcal{S}_{KeY}$, this rule is not correct.

6 A Sequent Calculus for Java Integer Arithmetic

In this chapter, we present a sequent calculus for JAVA integer arithmetic based on the semantics $\mathcal{S}_{KeY}$. In Section 6.1, we first give a grammar of the arithmetical expressions our calculus can handle.

6.1 Grammar of Java Integer Arithmetic

In this section, we give a grammar for JAVA integer arithmetic. For each expression that is derivable with the production of the grammar shown in Table 6.1, there exists one or possibly several rules in the calculus (which are referenced in the table) applicable to the expression. Note, that the referred rules are not automatically applicable without checking the requirements stated in the definition of the according rules.

The grammar in Table 6.1 is presented in the same style as the grammar for the whole JAVA language in chapter 18 of the JAVA language specification [Bracha et al., 2000]. But in contrast to the grammar in the JAVA language specification which is optimised for parsing, our grammar is easier to understand. Terminal symbols are written in type writer font and non-terminal symbols are written in italic font. As usual in the definition of grammars, T^+ means one or arbitrary many occurrences of the symbol T, T^* means zero or arbitrary many occurrences of T, and $S \mid T$ means either S or T. To keep the grammar concise, it contains no productions for the non-terminal symbols Id (identifier), $methodCall$, $IntegerLiteral$, and $LongLiteral$ but even without the productions for these symbols, their meaning should be clear.[1]

Production	Possibly applicable rules
$ExprStatement$`;`	
$StatementExpr$	
$StatementExpr$	
lhs `=` $nonSimpleExpr$	Rule 6.3.3
lhs `=` $srhs$	Rules 6.3.3 or 6.4.1
lhs $binOp$`=` $(nonSimpleExpr \mid srhs)$	Rules 6.3.3 or 6.3.5
lhs `=` $iteratedAssignment$	Rules 6.3.3 or 6.3.4
to be continued on the next page...	

[1] In doubt, we refer to the JAVA language specification.

...continuation of the table from the previous page	
Production	**Possibly applicable rules**
lhs `=` $incDecOp\ lhs$	Rules 6.6.1 or 6.6.2
lhs `=` $lhs\ incDecOp$	Rules 6.6.5 or 6.6.6
$incDecOp\ lhs$	Rules 6.6.3 or 6.6.4
$lhs\ incDecOp$	Rules 6.6.7 or 6.6.8
$booleanExpr$	
$(nonSimpleExpr \mid srhs)\ compOp$ $(nonSimpleExpr \mid srhs)$	Rules 6.3.6, 6.8.2, 6.8.3, or 6.8.4
$iteratedAssignment$	
$(\texttt{=}\ lhs)^{+}\ \texttt{=}\ \ nonSimpleExpr$	Rule 6.3.4
$nonSimpleExpr$	
$StatementExpr$	
$booleanExpr$	
$nonSimpleArrayAccess$	Rules 6.3.2
$\texttt{(}(nonSimpleExpr \mid srhs)\texttt{)}$	Rule 6.3.1
$\texttt{(}primType\texttt{)}(nonSimpleExpr \mid srhs)$	Rules 6.7.1, 6.7.2, 6.7.3, 6.7.4, or 6.7.5
$unaryOp\ (nonSimpleExpr \mid srhs)$	Rules 6.3.7, 6.5.1, 6.5.2, 6.5.4, or 6.5.5
$(nonSimpleExpr \mid srhs)\ binOp$ $(nonSimpleExpr \mid srhs)$	Rules 6.3.6, 6.4.2, 6.4.3, or 6.4.5
$booleanExpr\ \texttt{?}\ (nonSimpleExpr \mid srhs)\ \texttt{:}$ $(nonSimpleExpr \mid srhs)$	Rule 6.8.1
lhs	
$slhs$	
$nonSimpleArrayAccess$	Rule 6.3.3
$slhs$	
$nonArrayAccess$	
$simpleArrayAccess$	
$nonArrayAccess$	
$localVar$	
$instanceVar$	
$staticVar$	
$simpleArrayAccess$	
$Id(.Id)^{*}(\texttt{[}nonArrayAccess \mid literal\texttt{]})^{+}$	
$nonSimpleArrayAccess$	
$Id(.Id)^{*}(\texttt{[}StatementExpr \mid srhs\texttt{]})^{+} \mid$ $methodCall(\texttt{[}StatementExpr \mid srhs\texttt{]})^{+}$	Rule 6.3.2
$srhs$	
$slhs$	
$literal$	
to be continued on the next page...	

<table>
<tr><td colspan="2">...continuation of the table from the previous page</td></tr>
<tr><th>Production</th><th>Possibly applicable rules</th></tr>
<tr><td>primType</td><td></td></tr>
<tr><td>   byte</td><td></td></tr>
<tr><td>   short</td><td></td></tr>
<tr><td>   int</td><td></td></tr>
<tr><td>   long</td><td></td></tr>
<tr><td>   arithByte</td><td></td></tr>
<tr><td>   arithShort</td><td></td></tr>
<tr><td>   arithInt</td><td></td></tr>
<tr><td>   arithLong</td><td></td></tr>
<tr><td>binOp</td><td></td></tr>
<tr><td>   + | - | * | / | % | >> | >>> |</td><td></td></tr>
<tr><td>   << | & | ^ | | | && | ||</td><td></td></tr>
<tr><td>unaryOp</td><td></td></tr>
<tr><td>   +</td><td></td></tr>
<tr><td>   -</td><td></td></tr>
<tr><td>   ~</td><td></td></tr>
<tr><td>incDecOp</td><td></td></tr>
<tr><td>   ++</td><td></td></tr>
<tr><td>   --</td><td></td></tr>
<tr><td>compOp</td><td></td></tr>
<tr><td>   ==</td><td></td></tr>
<tr><td>   !=</td><td></td></tr>
<tr><td>   <</td><td></td></tr>
<tr><td>   <=</td><td></td></tr>
<tr><td>   ></td><td></td></tr>
<tr><td>   >=</td><td></td></tr>
<tr><td>literal</td><td></td></tr>
<tr><td>   IntegerLiteral</td><td></td></tr>
<tr><td>   LongLiteral</td><td></td></tr>
</table>

Table 6.1: This table shows the grammar for JAVA integer arithmetic and the set of rules possibly applicable to the according production.

6.2 Definitions

In the following definition, we define the meta variables used in the rules of the calculus presented in this chapter.

Definition 6.2.1 (Meta variables)
The meta variables `slhs`*,* `srhs`*, and* `expr` *are defined as follows.*

Any word, that is derivable in the grammar shown in Table 6.1 with the start symbol

- *slhs is a simple left-hand side* `slhs`*.*
- *srhs is a simple right-hand side* `srhs`*.*
- *StatementExpr is an expression* `expr`*.*

`slhs` and `srhs` are defined such that their evaluation does not change their value and thus, they are called "simple". However, their evaluation is not free of side-effects because the evaluation of an attribute or an array access may cause run-time exceptions (e. g. a `NullPointerException` or an `IndexOutOfBoundsException`). Note, that `slhs` and `srhs` are also logical terms and as mentioned in Remark 2.2.19, their semantics is not defined if their evaluation in JAVA causes an exception. To ensure that no undefined terms occur, the rules of the calculus have premisses that check the definedness of `slhs` and `srhs` by means of the predicate *isdef*.

Many rules require the introduction of a "new" variable of a certain type which may depend on the type of an expression. Since we are only concerned with primitive types, the type of an expression can always be determined at compile time.

Definition 6.2.2 ("New" variable)
If a constraint is attached to a rule requiring the introduction of a "new" variable*, this means that the variable has to be new with respect to the conclusion of the rule.*

If a "new" variable is used in a program, we assume that the variable is already declared and thus, in the rules of the calculus new variables are not declared explicitly.

In the rules of the calculus we use the short forms from the following definition.

Definition 6.2.3 (Short forms in programs)
The Greek letters π, ω, and μ are used as short forms in JAVA *programs and are defined as follows.*

- *π and ω are program parts such that π contains only braces, labels,* `try` *statements,* `catch` *statements,* `return` *statements, and method calls and $\pi\ \omega$ is a correct program.*
- *μ is a possibly empty set of updates attached to a formula.*

The rules of the calculus operate on the *first active statement* p of a program $\pi\ p\ \omega$. The prefix π is needed to keep track of the block the statement p belongs to, such that the statements `throw`, `return`, `break`, and `continue`, that change the control flow of the program, can be handled appropriately.[2]

To illustrate the definition of the short forms π, ω, and μ, we give a short example where the first active statement is labelled with p.

Example 6.2.4

$$\underbrace{\{\mathtt{j} \leftarrow 0\}}_{\mu}\underbrace{\mathtt{try\{}}_{\pi}\underbrace{\mathtt{int\ i=0;}}_{p}\ \underbrace{\mathtt{j++;\}\ catch\ (Exception\ e)\ \{\}}}_{\omega}$$

□

Now we have given all necessary definitions and in the following sections we present the rules of our sequent calculus for JAVA integer arithmetic. Basically, the calculus contains two kinds of rules. The first one are program transformation rules and the the second one are rules that symbolically execute a JAVA expression or statement. Program transformation rules apply if the arguments of an operator or the left-hand side of an assignment are not simple, this means their evaluation may have side-effects.[3] They are applied as often until all the expressions in the first active statements are simple and then, a rule for symbolic execution can be applied.

Remark 6.2.5
The most program transformation rules do not contain a premiss of the form

$$\Gamma \ \vdash \ isdef(\mathtt{slhs}),\ \Delta$$

which ensures the definedness of the access to `slhs`*. But for some of the program transformation rules, in which the evaluation order in the* JAVA *programs in the premiss and conclusion differ, such a premiss is necessary. For example, regard Rule 6.3.6*

$$\frac{\begin{array}{l} 1.\ \ \Gamma \ \vdash \ isdef(\mathtt{slhs}),\ \Delta \\ 2.\ \ \Gamma \ \vdash \ \mu\langle\pi\ \mathtt{y = expr_1;\ \ z = expr_2;\ \ \ slhs = y\ op\ z;}\ \ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \ \vdash \ \mu\langle\pi\ \mathtt{slhs = expr_1\ op\ expr_2;}\ \ \omega\rangle\phi,\ \Delta}$$

which transforms a compound binary expression statement into a sequence of simpler statements.

[2] Rules handling the statements `throw`, `return`, `break`, and `continue` can be found in [Beckert, 2001].

[3] As mentioned above, the evaluation of simple expressions is not free of side-effects but the possible side-effects (which are only exceptions that may be thrown) do not change their values.

In case that the evaluation of $\texttt{slhs}$ *terminates abruptly, in the program contained in the conclusion,* $expr_1$ *and* $expr_2$ *are not evaluated. In the program contained in the premiss, the expressions* $expr_1$ *and* $expr_2$ *are evaluated before* $\texttt{slhs}$ *is evaluated. Thus, if the evaluation of* $expr_1$ *or* $expr_2$ *has side-effects and the evaluation of* $\texttt{slhs}$ *terminates abruptly, the programs in the premiss and in the conclusion are not equivalent in general. Therefore, the definedness of the access to* $\texttt{slhs}$ *has to be ensured by the first premiss.*

6.3 Rules for Simplifying Compound Expressions

The rules in this section simplify assignment statements by removing redundant parentheses or by transforming them into a sequence of simpler statements.

Rule 6.3.1 *parentheses-elimination*

Purpose Elimination of redundant parentheses.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \Gamma \vdash \mu\langle\pi\ \ \texttt{slhs = } expr\texttt{;}\ \ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle\pi\ \ \texttt{slhs = (} expr\texttt{);}\ \ \omega\rangle\phi,\ \Delta}$$

Description This rule transforms a statement containing a pair of redundant parentheses into a semantically equivalent statement without the redundant parenthesis.

Rule 6.3.2 *array-access-simplification*

Purpose Simplification of a non-simple array access.

Requirements

- At least one of $expr_1 \ldots expr_n$ must not be a simple expression or a is not an identifier but a method invocation.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l}1.\ \Gamma \vdash isdef(\texttt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mu\langle\pi\ b\ \texttt{=}\ a\texttt{;}\ x_1\ \texttt{=}\ expr_1\texttt{;}\ \ \ldots\ x_n\ \texttt{=}\ expr_n\texttt{;} \\ \qquad\qquad \texttt{slhs}\ \texttt{=}\ b\texttt{[}x_1\texttt{]}\ldots\texttt{[}x_n\texttt{]}\texttt{;}\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mu\langle\pi\ \texttt{slhs}\ \texttt{=}\ a\texttt{[}expr_1\texttt{]}\ \ldots\ \texttt{[}expr_n\texttt{];}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- a is an array reference type which may be a variable of an array type or an invocation of a method with an array type as return type.
- b is a new variable of the type of a.
- x_i is a new variable of the type of $expr_i$.

This rules transforms a non-simple array access into a simple array access. This means, after the transformation, the array indices are simple expressions and the array reference is a variable.

According to the JAVA language specification, on the right-hand side of the assignment, at first the array reference `a` is evaluated and if it terminates abruptly, the evaluation of the whole array access terminates abruptly and the indices are not evaluated. To preserve this evaluation order in the program in the premiss, at first the array reference is assigned to a new variable `b` of the same type.

Rule 6.3.3 *left-hand-side-array-assignment-simplification*

Purpose Simplification of a non-simple left-hand side array access.

Requirements

- At least one of $expr_1 \ldots expr_n$ must not be a simple expression or a is not an identifier but a method call.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \Gamma \vdash \mu\langle\pi\ b = a;\ x_1 = expr_1;\ \ldots\ x_n = expr_n;\quad b[x_1]\ldots[x_n]\ op\ expr;\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle\pi\ a[expr_1]\ \ldots\ [expr_n]\ op\ expr;\ \omega\rangle\phi,\ \Delta}$$

Description

- a is an array reference type which may be a variable of an array type or an invocation of a method with an array type as return type.
- b is a new variable of the type of a.
- x_i is a new variable of the type of $expr_i$.
- op is one of the assignment operators `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `|=`, `^=`.

This rules transforms a non-simple array access on the left-hand side of an assignment into a simple array access. This means, after the transformation, the array indices are simple expressions and the array reference is a variable.

Rule 6.3.4 *iterated-assignment-simplification*

Purpose Simplification of iterated assignments.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \ \Gamma \ \vdash \ \mu\langle\pi\ \mathtt{\mathit{slhs}_2\ =\ \mathit{expr};\ \ \mathit{slhs}_1\ =\ \mathit{slhs}_2;}\ \omega\rangle\phi,\ \Delta}{\Gamma \ \vdash \ \mu\langle\pi\ \mathtt{\mathit{slhs}_1\ =\ \mathit{slhs}_2\ =\ \mathit{expr};}\ \omega\rangle\phi,\ \Delta}$$

Description This rule transforms an iterated assignment statement into a sequence of assignment statements of less complexity. Since the assignment operator is right-associative, the assignment $\mathtt{\mathit{slhs}_2 = \mathit{expr};}$ has to be evaluated first.

Rule 6.3.5 *compound-assignment-operator-transformation*

Purpose Transformation of a statement containing compound assignment operators into a semantically equivalent statement containing only the simple assignment operator.

Requirements

- The left-hand side *slhs* of the compound assignment operator has to be of a primitive type.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \ \Gamma \ \vdash \ \mu\langle\pi\ \mathtt{\mathit{slhs}\ =\ (\mathit{T})(\mathit{slhs}\ \mathit{op}\ \mathit{expr});}\ \ \omega\rangle\phi,\ \Delta}{\Gamma \ \vdash \ \mu\langle\pi\ \mathtt{\mathit{slhs}\ \mathit{op}\texttt{=}\ \mathit{expr}\ ;}\ \omega\rangle\phi,\ \Delta}$$

Description

- *op* is one of the binary operators `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `|`, `^`.
- *T* is the type of the *slhs* .

This rule does not symbolically execute the compound assignment statement but transforms it into a semantically equivalent statement containing the simple assignment operator.

It is important that the left-hand side of the compound assignment operator is a simple left-hand side *slhs* and thus has has no side-effects changing its value. This ensures the semantically equivalence between the programs in the premiss and in the conclusion because evaluating the statement `slhs = (T)(slhs op expr);` , *slhs* is evaluated twice in contrast to the statement `slhs op= expr;` where *slhs* is only evaluated once.

Rule 6.3.6 *compound-binary-expression-simplification*

Purpose Simplification of a statement containing compound arithmetical expressions.

Requirements

- At least one of expr_1 and expr_2 is not a simple expression.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l}1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mu\langle\pi\ \mathtt{y = }\mathit{expr}_1\mathtt{;}\ \ \mathtt{z = }\mathit{expr}_2\mathtt{;}\ \ \mathtt{slhs = y}\ \mathit{op}\ \mathtt{z;}\ \ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs = }\mathit{expr}_1\,\mathit{op}\,\mathit{expr}_2\mathtt{;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- *op* is one of the binary arithmetical operator `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^`, `|` or one of the comparison operators `<`, `>`, `<=`, `>=`, `==`, `!=`.
- `y` is a new variable of type of the expression expr_1.
- `z` is a new variable of type of the expression expr_2.

This rule transforms a statement containing a compound binary expression into a semantically equivalent sequence of statements in which at first the compound expressions are evaluated. The results of the evaluations are stored in new variables and finally, the binary operator is applied to the new variables.

The requirement, that at least one of both arguments has to be a compound expression, ensures that this rule is not applicable arbitrary often. Applying this rule more than one time does not contribute to further simplification of the program and might result in a loop when applying this rule automatically.

Rule 6.3.7 *compound-unary-expression-simplification*

Purpose Simplification of a statement containing an unary operator applied to a compound expression.

Requirements

- *expr* is not a simple expression.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l}1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mu\langle\pi\ \mathtt{y = }\mathit{expr}\mathtt{;}\ \ \mathtt{slhs = }\mathit{op}\ \mathtt{y;}\ \ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs = }\mathit{op}\ \mathit{expr}\mathtt{;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- *op* is one of the unary arithmetical operators `+`, `-`, `~` or the unary boolean negation operator `!`.
- *y* is a new variable of type of the expression *expr*.

This rule transforms a statement containing an unary operator applied to a compound expression into a semantically equivalent sequence of statements in which at first the compound expression is evaluated. The result of this evaluation is stored in a new variable and finally, the unary operator is applied to the new variable.

As in Rule 6.3.6, the requirement, that the expression *expr* is not allowed to be a simple expression, ensures that this rule is not applicable arbitrary often.

6.4 Rules for Handling Simple Expressions

In contrast to the previous section, the rules in this section actually symbolically execute statements. This requires accessing the values of variables and attributes which may cause run-time exceptions if the access is not defined. For example, a `NullPointerException` is thrown trying to access the value `o.a` of the attribute `a` of object `o` if the evaluation of `o` yields `null`. To ensure that the access to variables and attributes is defined, the rules have additional premisses of the form $\vdash$ *isdef*(*srhs*) which is valid if and only if the access to the value of *srhs* is defined.

Rule 6.4.1 *assignment-statement*

Purpose Symbolic execution of an assignment statement.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\ 3.\ \Gamma \vdash (\{\mathtt{slhs} \leftarrow \mathtt{srhs}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs\ =\ srhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description This rules symbolically executes an assignment statement. The symbolic execution results in an update attached to the formula that contained the assignment statement before. This means, after the symbolic execution of the assignment, the formula is evaluated in a state where *slhs* has the value *srhs* and the assignment has disappeared from the program.

Rule 6.4.2 *arithmetical-binary-expression-execution-right*

Purpose Symbolic execution of a binary expression of an arithmetical type for the diamond operator.

Requirements

- The type of the expression $\mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2$ has to be an arithmetical type.

1. $\Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta$
2. $\Gamma \vdash \mathit{isdef}(\mathtt{srhs}_1),\ \Delta$
3. $\Gamma \vdash \mathit{isdef}(\mathtt{srhs}_2),\ \Delta$
4. $\Gamma \vdash \neg \mathtt{srhs}_2 \doteq 0$
5. $\Gamma,\ \mathit{in}_{T_1}(\mathtt{srhs}_1),\ \mathit{in}_{T_2}(\mathtt{srhs}_2),\ \mathit{in}_{T}(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2) \vdash$
   $(\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta$
6. $\Gamma,\ \mathit{in}_{T_1}(\mathtt{srhs}_1),\ \mathit{in}_{T_2}(\mathtt{srhs}_2),\ \neg\mathit{in}_{T}(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2) \vdash$
   $\mu\langle \pi\ \mathtt{slhs}\ \mathtt{=}\ \mathtt{overflow(}\mathtt{srhs}_1\mathtt{,}\mathtt{srhs}_2\mathtt{,"}\mathtt{op}\mathtt{");}\ \ \omega\rangle\phi,\ \Delta$
7. $\Gamma,\ \neg(\mathit{in}_{T_1}(\mathtt{srhs}_1)\ \wedge \mathit{in}_{T_2}(\mathtt{srhs}_2)) \vdash$
   $(\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta$

$\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}\ \mathtt{=}\ \mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2\mathtt{;}\ \ \omega\rangle\phi,\ \Delta$

Description

- $\mathtt{op}$ is one of the binary operators `*`, `/`, `%`, `+`, `-`.
- $\mathtt{logOp}$ is the corresponding logical function to the JAVA operator $\mathtt{op}$:
  - If $\mathtt{op}$ is `*`, then $\mathtt{logOp}$ is $*$.
  - If $\mathtt{op}$ is `/`, then $\mathtt{logOp}$ is $/$.
  - If $\mathtt{op}$ is `%`, then $\mathtt{logOp}$ is *mod*.
  - If $\mathtt{op}$ is `+`, then $\mathtt{logOp}$ is $+$.
  - If $\mathtt{op}$ is `-`, then $\mathtt{logOp}$ is $-$.
- T_1 is the type of $\mathtt{srhs}_1$.
- T_2 is the type of $\mathtt{srhs}_2$.
- T is the type of $\mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2$.

This rule symbolically executes an assignment statement where the right-hand side is a binary expression of an arithmetical type.

The gray shaded premiss number four only applies if $\mathtt{op}$ is the division operator `/` or the remainder operator `%`, otherwise this premiss has to be ignored. It ensures that the divisor is not zero which would cause an `ArithmeticException`. If the divisor is zero, Rule 6.4.4 has to be applied.

For the explanation of the premisses of the rule, we assume that Γ holds and Δ not (otherwise the rule is applicable trivially).

Premiss number five states that, if the arguments are in valid range and the result is in valid range as well (and thus no overflow occurs), the arithmetical JAVA operation is symbolically executed. This results in an update attached to the formula.

Premiss number six states that, if the arguments are in valid range but the result would be not, the result of the arithmetical JAVA operation is the return value of the method invocation of `overflow`. Since we are using semantics $\mathcal{S}_{KeY}$, we have no specification for the method `overflow` and thus, the formula

$$\mu\langle\pi\ \mathtt{slhs}\ \mathtt{=}\ \mathtt{overflow(}\mathtt{srhs}_1\mathtt{,}\mathtt{srhs}_2\mathtt{,"}\mathtt{op}\mathtt{");}\quad \omega\rangle\phi$$

is unsatisfiable. Therefore, this formula could be left out without changing the semantics of the rule. However, for certain variants of $\mathcal{S}_{KeY}$ (see Section 5.4) this formula may be satisfiable and thus, the presence of this formula allows to use a variant of $\mathcal{S}_{KeY}$ without having to change the rules of the calculus.

Premiss number seven is for the case that at least one argument is not in valid range. This can only happen in unreal states and thus, overflow is allowed since it does not affect the behaviour of the program during the execution on the virtual machine because unreal states are not reachable by the virtual machine.

An equivalent variant of this rule would be to replace the premisses number five and seven with the single premiss

$$\begin{gathered}\Gamma,\ in_T(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2)\ \vdash \\ (\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2\} \circ \mu)\langle\pi\ \omega\rangle\phi,\ \Delta.\end{gathered}$$

From the above sequent the other two sequents are derivable by applying cut and weakening rules.

Note, that *op* must be none of the bit operators `<<`, `>>`, `>>>`, `&`, `^`, and `|` because bit operators are not allowed to be applied to expressions of the arithmetical types.

Rule 6.4.3 *arithmetical-binary-expression-execution-left*

Purpose Symbolic execution of a binary expression of an arithmetical type for the box operator.

Requirements

- The type of the expression $\mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2$ has to be an arithmetical type.

$$
\begin{array}{l}
1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\
2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_1),\ \Delta \\
3.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_2),\ \Delta \\
4.\ \Gamma \vdash \neg \mathtt{srhs}_2 \doteq 0 \\
5.\ \Gamma,\ in_{T_1}(\mathtt{srhs}_1),\ in_{T_2}(\mathtt{srhs}_2),\ in_T(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2) \vdash \\
\qquad (\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2\} \circ \mu)[\pi\ \omega]\phi,\ \Delta \\
6.\ \Gamma,\ in_{T_1}(\mathtt{srhs}_1),\ in_{T_2}(\mathtt{srhs}_2),\ \neg in_T(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2) \vdash \\
\qquad \mu[\pi\ \mathtt{slhs}\ \texttt{= overflow(}\mathtt{srhs}_1\texttt{,}\mathtt{srhs}_2\texttt{,"}\mathtt{op}\texttt{");}\ \ \omega]\phi,\ \Delta \\
7.\ \Gamma,\ \neg(in_{T_1}(\mathtt{srhs}_1)\ \wedge in_{T_2}(\mathtt{srhs}_2)),\ \vdash \\
\qquad (\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2\} \circ \mu)[\pi\ \omega]\phi,\ \Delta \\
\hline
\Gamma \vdash \mu[\pi\ \mathtt{slhs}\ \texttt{=}\ \mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2;\ \ \omega]\phi,\ \Delta
\end{array}
$$

Description

- *op* is one of the binary operators `*`, `/`, `%`, `+`, `-`.
- *logOp* is the corresponding logical function to the Java operator *op*:
  - If *op* is `*`, then *logOp* is $*$.
  - If *op* is `/`, then *logOp* is $/$.
  - If *op* is `%`, then *logOp* is *mod*.
  - If *op* is `+`, then *logOp* is $+$.
  - If *op* is `-`, then *logOp* is $-$.
- T_1 is the type of $\mathtt{srhs}_1$.
- T_2 is the type of $\mathtt{srhs}_2$.
- T is the type of $\mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2$.

This rule symbolically executes an assignment statement where the right-hand side is a binary expression of an arithmetical type.

The gray shaded premiss number four only applies if *op* is the division operator `/` or the remainder operator `%`, otherwise this premiss has to be ignored. It ensures that the divisor is not zero which would cause an `ArithmeticException`. If the divisor is zero, Rule 6.4.4 has to be applied.

For the explanation of the premisses of the rule, we assume that Γ holds and Δ not (otherwise the rule is applicable trivially).

Premiss number five states that, if the arguments are in valid range and the result is in valid range as well (and thus no overflow occurs), the arithmetical Java operation is symbolically executed. This results in an update attached to the formula.

Premiss number six states that, if the arguments are in valid range but the result would be not, the result of the arithmetical JAVA operation is the return value of the method invocation of `overflow`. Note, that in contrast to Rule 6.4.2, the formula

$$\mu[\pi\ \mathtt{slhs}\ \texttt{= overflow(}\mathtt{srhs}_1\texttt{,}\mathtt{srhs}_2\texttt{," }\mathtt{op}\texttt{");}\ \ \omega]\phi$$

is satisfiable in $\mathcal{S}_{KeY}$ if ϕ evaluates to *true* (in this case ϕ is even valid) and therefore may not be left out.

Premiss number seven is for the case that at least one argument is not in valid range. This can only happen in unreal states and thus, overflow is allowed since it does not affect the behaviour of the program during the execution on the virtual machine because unreal states are not reachable by the virtual machine.

An equivalent variant of this rule would be to replace the premisses number five and seven with the single premiss

$$\Gamma,\ in_T(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2) \vdash$$
$$(\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2\} \circ \mu)[\pi\ \omega]\phi,\ \Delta.$$

From the above sequent the other two sequents are derivable by applying cut and weakening rules.

Note, that *op* must be none of the bit operators `<<`, `>>`, `>>>`, `&`, `^`, and `|` because bit operators are not allowed to be applied to expressions of the arithmetical types.

Rule 6.4.4 *division-by-zero*

Purpose Symbolic execution of a division by zero.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{ll}1. & \Gamma \vdash isdef(\mathtt{slhs}),\ \Delta \\ 2. & \Gamma \vdash isdef(\mathtt{srhs}_1),\ \Delta \\ 3. & \Gamma \vdash isdef(\mathtt{srhs}_2),\ \Delta \\ 4. & \Gamma \vdash \mathtt{srhs}_2 \doteq 0 \\ 5. & \Gamma \vdash \mu\langle\pi\ \texttt{throw new ArithmeticException();}\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}\ \texttt{=}\ \mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2\texttt{;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- $\mathtt{op}$ is the division operator `/` or the remainder operator `%`.
- $\mathtt{logOp}$ is the corresponding logical function to the JAVA operator $\mathtt{op}$:
  - If $\mathtt{op}$ is `/`, then $\mathtt{logOp}$ is $/$.
  - If $\mathtt{op}$ is `%`, then $\mathtt{logOp}$ is mod.
- T_1 is the type of $\mathtt{srhs}_1$.
- T_2 is the type of $\mathtt{srhs}_2$.

This rule symbolically executes a division or remainder operation with the divisor being zero and thus, an `ArithmeticException` is thrown.

Rule 6.4.5 *built-in-binary-expression-execution*

Purpose Symbolic execution of a binary expression of a built-in type.

Requirements

- The type of the expression $\mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2$ has to be a built-in type.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{ll} 1. & \Gamma \vdash isdef(\mathtt{slhs}),\ \Delta \\ 2. & \Gamma \vdash isdef(\mathtt{srhs}_1),\ \Delta \\ 3. & \Gamma \vdash isdef(\mathtt{srhs}_2),\ \Delta \\ 4. & \Gamma \vdash \neg \mathtt{srhs}_2 \doteq 0 \\ 5. & \Gamma \vdash (\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2\} \circ \mu)\langle \pi\ \omega \rangle \phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}\ \mathtt{=}\ \mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2\mathtt{;}\ \ \omega \rangle \phi,\ \Delta}$$

Description

- $\mathtt{op}$ is one of the binary operators `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `|`, and `^`.
- $\mathtt{logOp}$ is the corresponding logical function to the JAVA operator $\mathtt{op}$:
  - If $\mathtt{op}$ is `*`, then $\mathtt{logOp}$ is $JMul_T$.
  - If $\mathtt{op}$ is `/`, then $\mathtt{logOp}$ is $JDiv_T$.
  - If $\mathtt{op}$ is `%`, then $\mathtt{logOp}$ is $JMod_T$.
  - If $\mathtt{op}$ is `+`, then $\mathtt{logOp}$ is $JPlus_T$.
  - If $\mathtt{op}$ is `-`, then $\mathtt{logOp}$ is $JMinus_T$.
  - If $\mathtt{op}$ is `<<`, then $\mathtt{logOp}$ is $JShiftLeft_T$.
  - If $\mathtt{op}$ is `>>`, then $\mathtt{logOp}$ is $JShiftRight_T$.

- If op is `>>>`, then $logOp$ is $JUnsigned\ ShiftRight_T$.
- If op is `&`, then $logOp$ is $JAnd_T$.
- If op is `|`, then $logOp$ is JOr_T.
- If op is `^`, then $logOp$ is $JXor_T$.

- T_1 is the type of $srhs_1$.
- T_2 is the type of $srhs_2$.
- T is the type of $srhs_1\ op\ srhs_2$.

This rule symbolically executes an assignment statement where the right-hand side is a binary expression of a built-in type.

The gray shaded premiss number four only applies if op is the division operator `/` or the remainder operator `%`, otherwise this premiss has to be ignored. It ensures that the divisor is not zero which would cause an `ArithmeticException`. If the divisor is zero, Rule 6.4.4 has to be applied.

6.5 Rules for Handling Unary Operators

The rules in this section symbolically execute the unary plus operator `+`, the unary minus operator `-`, and the bitwise complement operator `~`.

Rule 6.5.1 *unary-plus*

Purpose Symbolic execution of the unary plus operator applied to a simple expression.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash isdef(slhs),\ \Delta \\ 2.\ \Gamma \vdash isdef(srhs),\ \Delta \\ 3.\ \Gamma \vdash (\{slhs \leftarrow srhs\} \circ \mu)\langle \pi\ \omega \rangle \phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ slhs\ \texttt{=}\ \texttt{+}srhs\texttt{;}\ \ \omega \rangle \phi,\ \Delta}$$

Description This rule symbolically executes the unary plus operator. This actually means just to remove the operator because the unary plus operator does not change the value of the argument and a unary plus expression has no side-effects.

In contrast to the unary minus operator, there is only one rule for the unary plus operator as well for the built-in types as for the arithmetical types since the unary plus operator cannot cause overflow.

Rule 6.5.2 *arithmetical-unary-minus-right*

Purpose Symbolic execution of the unary minus operator applied to a simple expression of an arithmetical type for the diamond operator.

Requirements
- The type of $\mathtt{srhs}$ has to be an arithmetical type.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\ 3.\ \Gamma,\ \mathtt{srhs} \doteq \mathtt{MIN\_}T \vdash \\ \quad \mu\langle\pi\ \mathtt{slhs = overflow(srhs,"-");}\ \ \omega\rangle\phi,\ \Delta \\ 4.\ \Gamma,\ \neg\mathtt{srhs} \doteq \mathtt{MIN\_}T \vdash (\{\mathtt{slhs} \leftarrow -\mathtt{srhs}\} \circ \mu)\langle\pi\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs = -srhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- T is the promoted type of $\mathtt{srhs}$.
- $\mathtt{MIN\_}T$ is the minimum value of the corresponding built-in type of the arithmetical type T.

This rule symbolically executes the assignment of an unary minus expression of an arithmetical type.

To explain the premisses, we assume that Γ holds and Δ not (otherwise the rule is applicable trivially).

The third premiss applies to the case that $\mathtt{srhs}$ is equal to the minimum value $\mathtt{MIN\_}T$ of the corresponding built-in type of T. In this case, as described in Section 4.2.6, overflow occurs. Thus, the result of the unary minus expression is the return value of the method invocation of `overflow`.

The fourth premiss states that the result of the unary minus expression is the mathematical negation of the value of $\mathtt{srhs}$, if $\mathtt{srhs}$ is different from $\mathtt{MIN\_}T$. In this case, in real states no overflow occurs. In unreal states overflow occurs but is allowed since unreal states are not reachable by the JAVA virtual machine.

Rule 6.5.3 *arithmetical-unary-minus-left*

Purpose Symbolic execution of the unary minus operator applied to a simple expression of an arithmetical type for the box operator.

Requirements
- The type of $\mathtt{srhs}$ has to be an arithmetical type.

1. $\Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta$
2. $\Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta$
3. $\Gamma,\ \mathtt{srhs} \doteq \mathtt{MIN\_}T \vdash$
   $\mu[\pi\ \mathtt{slhs}\ \texttt{= overflow(}\mathtt{srhs}\texttt{,"-");}\ \ \omega]\phi,\ \Delta$
4. $\Gamma,\ \neg\mathtt{srhs} \doteq \mathtt{MIN\_}T \vdash (\{\mathtt{slhs} \leftarrow -\mathtt{srhs}\} \circ \mu)[\pi\ \omega]\phi,\ \Delta$

$$\frac{}{\Gamma \vdash \mu[\pi\ \mathtt{slhs}\ \texttt{= -}\mathtt{srhs}\texttt{;}\ \ \omega]\phi,\ \Delta}$$

Description

- T is the promoted type of $\mathtt{srhs}$.
- $\mathtt{MIN\_}T$ is the minimum value of the corresponding built-in type of the arithmetical type T.

This rule symbolically executes the assignment of an unary minus expression of an arithmetical type.

To explain the premisses, we assume that Γ holds and Δ not (otherwise the rule is applicable trivially).

The third premiss applies to the case that $\mathtt{srhs}$ is equal to the minimum value $\mathtt{MIN\_}T$ of the corresponding built-in type of T. In this case, as described in Section 4.2.6, overflow occurs. Thus, the result of the unary minus expression is the return value of the method invocation of `overflow`.

The fourth premiss states that the result of the unary minus expression is the mathematical negation of the value of $\mathtt{srhs}$, if $\mathtt{srhs}$ is different from $\mathtt{MIN\_}T$. In this case, in real states no overflow occurs. In unreal states overflow occurs but is allowed since unreal states are not reachable by the JAVA virtual machine.

Rule 6.5.4 *built-in-unary-minus*

Purpose Symbolic execution of the unary minus operator applied to a simple expression of a built-in type.

Requirements

- The type of $\mathtt{srhs}$ has to be a built-in type.

Remark The rule for the box operator $[\cdot]$ is analogous.

1. $\Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta$
2. $\Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta$
3. $\Gamma,\ \mathtt{srhs} \doteq \mathtt{MIN\_}T \vdash (\{\mathtt{slhs} \leftarrow \mathtt{srhs}\} \circ \mu)\langle\pi\ \omega\rangle\phi,\ \Delta$
4. $\Gamma,\ \neg\mathtt{srhs} \doteq \mathtt{MIN\_}T \vdash (\{\mathtt{slhs} \leftarrow -\mathtt{srhs}\} \circ \mu)\langle\pi\ \omega\rangle\phi,\ \Delta$

$$\frac{}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}\ \texttt{= -}\mathtt{srhs}\texttt{;}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- T is the promoted type of *srhs*.
- MIN_T is the minimum value of type T.

To explain the premisses, we assume that Γ holds and Δ not (otherwise the rule is applicable trivially).

The third premiss applies to the case that *srhs* is equal to the minimum value MIN_T of type T. In this case, as described in Section 4.2.6, overflow occurs and the result value of the unary minus expression is the same as the value of *srhs*.

The fourth premiss states that the result of the unary minus expression is the mathematical negation of the value of *srhs*, if *srhs* is different from MIN_T, because in this case no overflow occurs.

Rule 6.5.5 *built-in-bitwise-complement*

Purpose Symbolic execution of the bitwise complement operator.

Requirements
- The type of *srhs* has to be a built-in type.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\ 3.\ \Gamma \vdash (\{\mathtt{slhs} \leftarrow \mathit{JComplement}_T(\mathtt{srhs})\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs\ =\tilde{}srhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- T is the promoted type of *srhs*.
- MIN_T is the minimum value of type T.

This rule symbolically executes the bitwise complement operator applied to a simple expression of a built-in type. Note, that bit operators are not allowed to be applied to arithmetical types.

6.6 Rules for Handling Increment and Decrement Expressions

The rules in this section handle increment and decrement expressions (both prefix and postfix) by transforming them into binary addition or subtraction expressions. To guarantee that the original expression and the transformed expression are semantically equivalent, the type cast which is implicitly performed evaluating an increment or decrement expression has to be made explicit in the transformed expression.

Note, that in contrast to the other rules, the increment and decrement operators are applied to a simple left-hand side instead of a simple right-hand. The reason for that is, that these operators are not allowed to be applied to literals (which are simple right-hand sides) because the argument of the operator must evaluate to a variable.

Rule 6.6.1 *prefix-increment-expression*

Purpose Transformation of a prefix increment expression into a JAVA addition.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mu\langle \pi\ \mathtt{slhs}_2\ \mathtt{=}\ \mathtt{(}T\mathtt{)(}\mathtt{slhs}_2\ \mathtt{+}\ \mathtt{1);}\ \mathtt{slhs}_1\ \mathtt{=}\ \mathtt{slhs}_2\mathtt{;}\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}_1\ \mathtt{=}\ \mathtt{++}\mathtt{slhs}_2\mathtt{;}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- T is the type of $\mathtt{slhs}_1$.

Rule 6.6.2 *prefix-decrement-expression*

Purpose Transformation of a prefix decrement expression into a JAVA subtraction.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mu\langle \pi\ \mathtt{slhs}_2\ \mathtt{=}\ \mathtt{(}T\mathtt{)(}\mathtt{slhs}_2\ \mathtt{-}\ \mathtt{1);}\ \mathtt{slhs}_1\ \mathtt{=}\ \mathtt{slhs}_2\mathtt{;}\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}_1\ \mathtt{=}\ \mathtt{--}\mathtt{slhs}_2\mathtt{;}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- T is the type of $\mathtt{slhs}_1$.

Rule 6.6.3 *prefix-increment-statement*

Purpose Transformation of a prefix increment statement into a JAVA addition.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \Gamma \vdash \mu\langle\pi\ \mathtt{slhs = (T)(slhs + 1);}\ \ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle\pi\ \mathtt{{+}{+}slhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- $\mathtt{T}$ is the promoted type of $\mathtt{slhs}$.

Rule 6.6.4 *prefix-decrement-statement*

Purpose Transformation of a prefix decrement statement into a JAVA subtraction.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \Gamma \vdash \mu\langle\pi\ \mathtt{slhs = (T)(slhs - 1);}\ \ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle\pi\ \mathtt{{-}{-}slhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- $\mathtt{T}$ is the type of $\mathtt{slhs}$.

Rule 6.6.5 *postfix-increment-expression*

Purpose Transformation of a postfix increment expression into a JAVA addition.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \Gamma \vdash \mu\langle\pi\ \mathtt{slhs}_1\ \mathtt{=}\ \mathtt{slhs}_2\mathtt{;}\ \mathtt{slhs}_2\ \mathtt{= (T)(}\mathtt{slhs}_2\ \mathtt{+ 1);}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}_1\ \mathtt{=}\ \mathtt{slhs}_2\mathtt{{+}{+};}\ \ \omega\rangle\phi,\ \Delta}$$

Description
- $\mathtt{T}$ is the type of $\mathtt{slhs}_1$.

Rule 6.6.6 *postfix-decrement-expression*

Purpose Transformation of a postfix decrement expression into a JAVA subtraction.

Remark The rule for the box operator $[\cdot]$ is analogous.

1. $\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}_1 \texttt{ = } \mathtt{slhs}_2;\ \mathtt{slhs}_2 \texttt{ = (}T\texttt{)(}\mathtt{slhs}_2 \texttt{ - 1);}\ \omega\rangle\phi,\ \Delta$
$\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}_1 \texttt{ = } \mathtt{slhs}_2\texttt{--;}\ \ \omega\rangle\phi,\ \Delta$

Description
- T is the type of $\mathtt{slhs}_1$.

Rule 6.6.7 *postfix-increment-statement*

Purpose Transformation of a postfix increment statement into a JAVA addition.

Remark The rule for the box operator $[\cdot]$ is analogous.

1. $\Gamma \vdash \mu\langle\pi\ \mathtt{slhs} \texttt{ = (}T\texttt{)(}\mathtt{slhs} \texttt{ + 1);}\ \ \omega\rangle\phi,\ \Delta$
$\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}\texttt{++;}\ \ \omega\rangle\phi,\ \Delta$

Description
- T is the type of $\mathtt{slhs}$.

Rule 6.6.8 *postfix-decrement-statement*

Purpose Transformation of a postfix decrement statement into a JAVA subtraction.

Remark The rule for the box operator $[\cdot]$ is analogous.

1. $\Gamma \vdash \mu\langle\pi\ \mathtt{slhs} \texttt{ = (}T\texttt{)(}\mathtt{slhs} \texttt{ - 1);}\ \ \omega\rangle\phi,\ \Delta$
$\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}\texttt{--;}\ \ \omega\rangle\phi,\ \Delta$

Description
- T is the type of $\mathtt{slhs}$.

6.7 Rules for Handling Type Casts

The rules in this section handle type cast operations. Basically, there are three kinds of type casts, namely narrowing casts, identity casts, and widening casts. Note, that casts from an arithmetical type to a built-in type are always narrowing casts which may cause the invocation of the method `overflow` if overflow occurs.

Rule 6.7.1 *cast-simplification*

Purpose Simplification of a cast operator applied to a non-simple expression.

Requirements

- $\mathtt{expr}$ must not be a simple expression.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mu\langle\pi\ \mathtt{x = expr;\ slhs = (T)x;}\ \rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs = (T)expr;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- $\mathtt{x}$ is a new variable of the type of the expression $\mathtt{expr}$.

This rule simplifies an assignment statement containing a cast operator applied to a non-simple expression by transforming the assignment statement into two statements where at first the non-simple expression is evaluated and then assigned to a new variable. Finally, the cast operator is applied to this new variable.

Rule 6.7.2 *widening-or-identity-cast*

Purpose Symbolic execution of a widening or identity type cast.

Requirements

- For the type $\mathtt{S}$ of $\mathtt{srhs}$ holds that either $\mathtt{S}$ and the target type $\mathtt{T}$ of the cast operation are equal or $\mathtt{S} \stackrel{*}{\prec} \mathtt{T}$.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\ 3.\ \Gamma \vdash (\{\mathtt{slhs} \leftarrow \mathtt{srhs}\} \circ \mu)\langle\pi\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs = (T)srhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- $\mathtt{T}$ is the target type of the cast operation.

This rule symbolically executes a widening or identity type cast. Basically, widening and identity type cast cannot cause overflow and thus, there is only one rule for built-in types and arithmetical types.

Rule 6.7.3 *built-in-types-narrowing-cast*

Purpose Symbolic execution of a narrowing cast from a built-in type to another built-in type.

Requirements

- For the types S of srhs and the target type T of the cast operation holds $T \stackrel{*}{\prec} S$.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\ 3.\ \Gamma \vdash (\{\mathtt{slhs} \leftarrow \mathit{overflow}_T(\mathtt{srhs})\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs\ =\ (}T\mathtt{)srhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- T is the target type of the cast operation.

This rule symbolically executes a narrowing type cast from a built-in type to another built-in type.

The function $\mathit{overflow}_T$ is defined in Definition 2.2.14.

Rule 6.7.4 *arithmetical-narrowing-cast-right*

Purpose Symbolic execution of a narrowing cast from an arithmetical type to another arithmetical type or to a built-in type for the diamond operator.

Requirements

- The target type T of the cast operation has to be an arithmetical type.
- For the types S of srhs and T holds $T \stackrel{*}{\prec} S$.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\ 3.\ \Gamma,\ \mathit{in}_S(\mathtt{srhs}),\ \mathit{in}_T(\mathtt{srhs}) \vdash \\ \qquad (\{\mathtt{slhs} \leftarrow \mathtt{srhs}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\ 4.\ \Gamma,\ \mathit{in}_S(\mathtt{srhs}),\ \neg\mathit{in}_T(\mathtt{srhs}) \vdash \\ \qquad \mu\langle \pi\ \mathtt{slhs} = \mathtt{overflow(srhs\ ,\ "cast(}T\mathtt{)");}\ \ \omega\rangle\phi,\ \Delta \\ 5.\ \Gamma,\ \neg\mathit{in}_S(\mathtt{srhs}) \vdash (\{\mathtt{slhs} \leftarrow \mathtt{srhs}\} \circ \mu)\langle \pi\ \omega\rangle\phi \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs\ =\ (}T\mathtt{)srhs;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- S is the type of srhs.

- T is the target type of the cast operation.

For the explanation of the premisses of the rule, we assume that Γ holds and Δ not (otherwise the rule is applicable trivially).

Premiss number three states that the cast can be symbolically executed if *srhs* is in valid range as well of its own type S as of the target type T, because in this case no overflow occurs.

Premiss number four states that, if *srhs* is in valid range of its own type S but not of the target type T, the result of the cast operation is the return value of the method invocation of `overflow`. Since we are using semantics $\mathcal{S}_{KeY}$, we have no specification for the method `overflow` and thus, the formula

$$\mu\langle\pi \; \mathit{slhs} \texttt{ = overflow(}\mathit{srhs}\texttt{,"cast(}T\texttt{)");} \;\; \omega\rangle\phi$$

is unsatisfiable. Therefore, this formula could be left out without changing the semantics of the rule. However, for certain variants of $\mathcal{S}_{KeY}$ (see Section 5.4) this formula may be satisfiable and thus, the presence of this formula allows to use a variant of $\mathcal{S}_{KeY}$ without having to change the rules of the calculus.

Premiss number five applies to the case that *srhs* is not in valid range of its own type S. This can only happen in unreal states and thus, overflow is allowed since it does not affect the behaviour of the program during the execution on the virtual machine because unreal states are not reachable by the virtual machine.

An equivalent variant of this rule would be to replace the premisses number three and five by the single premiss

$$\Gamma, \; in_T(\mathit{srhs}) \; \vdash \; (\{\mathit{slhs} \leftarrow \mathit{srhs}\} \circ \mu)\langle\pi \; \omega\rangle\phi, \; \Delta.$$

From the above sequent the other two sequents are derivable by applying cut and weakening rules.

Rule 6.7.5 *arithmetical-narrowing-cast-left*

Purpose Symbolic execution of a narrowing cast from an arithmetical type to another arithmetical type or to a built-in type for the box operator.

Requirements

- The target type T of the cast operation has to be an arithmetical type.
- For the types S of *srhs* and T holds $T \overset{*}{\prec} S$.

1. $\Gamma \vdash isdef(\texttt{slhs}), \Delta$
2. $\Gamma \vdash isdef(\texttt{srhs}), \Delta$
3. $\Gamma, in_S(\texttt{srhs}), in_T(\texttt{srhs}) \vdash$
   $(\{\texttt{slhs} \leftarrow \texttt{srhs}\} \circ \mu)[\pi\ \omega]\phi, \Delta$
4. $\Gamma, in_S(\texttt{srhs}), \neg in_T(\texttt{srhs}) \vdash$
   $\mu[\pi\ \texttt{slhs = overflow(srhs , "cast(T)");}\ \ \omega]\phi, \Delta$
5. $\Gamma, \neg in_S(\texttt{srhs}) \vdash (\{\texttt{slhs} \leftarrow \texttt{srhs}\} \circ \mu)[\pi\ \omega]\phi$

$\Gamma \vdash \mu[\pi\ \texttt{slhs = (T)srhs;}\ \ \omega]\phi, \Delta$

Description

- S is the type of $\texttt{srhs}$.
- T is the target type of the cast operation.

For the explanation of the premisses of the rule, we assume that Γ holds and Δ not (otherwise the rule is applicable trivially).

Premiss number three states that the cast can be symbolically executed if $\texttt{srhs}$ is in valid range as well of its own type S as of the target type T, because in this case no overflow occurs.

Premiss number four states that, if $\texttt{srhs}$ is in valid range of its own type S but not of the target type T, the result of the cast operation is the return value of the method invocation of `overflow`. Note, that in premiss number four, in contrast to Rule 6.7.4, the formula

$$\mu[\pi\ \texttt{slhs = overflow(srhs ,"cast(T)");}\ \ \omega]\phi$$

is satisfiable in $\mathcal{S}_{KeY}$, for example if ϕ evaluates to *true*.

Premiss number five applies to the case that $\texttt{srhs}$ is not in valid range of its own type S. This can only happen in unreal states and thus, overflow is allowed since it does not affect the behaviour of the program during the execution on the virtual machine because unreal states are not reachable by the virtual machine.

An equivalent variant of this rule would be to replace the premisses number three and five by the single premiss

$$\Gamma, in_T(\texttt{srhs}) \vdash (\{\texttt{slhs} \leftarrow \texttt{srhs}\} \circ \mu)[\pi\ \omega]\phi, \Delta.$$

From the above sequent the other two sequents are derivable by applying cut and weakening rules.

6.8 Rules for Conditional Expression and Numeric Comparison Operators

The rules in this section handle conditional expressions and numeric comparison operators. The rule for conditional expressions performs program transformation whereas expressions containing numeric comparison operators are symbolically executed.

Rule 6.8.1 *conditional-expression*

Purpose Transformation of a conditional expression.

Requirements
- The type of expr is `boolean`.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \langle \pi\ \mathtt{if}\ (\mathit{expr})\ \mathtt{slhs}\ \mathtt{=}\ (T)\,\mathit{expr}_1; \\ \qquad\qquad \mathtt{else}\ \mathtt{slhs}\ \mathtt{=}\ (T)\,\mathit{expr}_2;\ \ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}\ \mathtt{=}\ \mathit{expr}\ \mathtt{?}\ \mathit{expr}_1\ \mathtt{:}\ \mathit{expr}_2;\ \ \omega\rangle\phi,\ \Delta}$$

Description
- T is the promoted type of $\mathit{expr}\ \mathtt{?}\ \mathit{expr}_1\ \mathtt{:}\ \mathit{expr}_2$.

This rules transforms a conditional expression into a if-else statement which ensures that only either expr_1 or expr_2 is evaluated, as it is in the evaluation of the conditional expression.

Rule 6.8.2 *equality-comparison*

Purpose Symbolic execution of an equality comparison.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l} 1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_1),\ \Delta \\ 3.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_2),\ \Delta \\ 4.\ \Gamma,\ \mathtt{srhs}_1 \doteq \mathtt{srhs}_2 \vdash (\{\mathtt{slhs} \leftarrow \mathit{true}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\ 5.\ \Gamma,\ \neg\mathtt{srhs}_1 \doteq \mathtt{srhs}_2 \vdash (\{\mathtt{slhs} \leftarrow \mathit{false}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}\ \mathtt{=}\ \mathtt{srhs}_1\mathtt{==}\mathtt{srhs}_2;\ \ \omega\rangle\phi,\ \Delta}$$

Description This rule symbolically executes an equality comparison of two simple expression.

Rule 6.8.3 *inequality-comparison*

Purpose Symbolic execution of an inequality comparison.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l}1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta\\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_1),\ \Delta\\ 3.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_2),\ \Delta\\ 4.\ \Gamma,\ \mathtt{srhs}_1 \doteq \mathtt{srhs}_2 \vdash (\{\mathtt{slhs} \leftarrow \mathit{false}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta\\ 5.\ \Gamma,\ \neg\mathtt{srhs}_1 \doteq \mathtt{srhs}_2 \vdash (\{\mathtt{slhs} \leftarrow \mathit{true}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs = srhs_1 != srhs_2;}\ \ \omega\rangle\phi,\ \Delta}$$

Description This rule symbolically executes an inequality comparison of two simple expression.

Rule 6.8.4 *numeric-comparison*

Purpose Symbolic execution of expressions containing comparison operators.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{\begin{array}{l}1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta\\ 2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_1),\ \Delta\\ 3.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_2),\ \Delta\\ 4.\ \Gamma,\ \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2 \vdash (\{\mathtt{slhs} \leftarrow \mathit{true}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta\\ 5.\ \Gamma,\ \neg(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2) \vdash (\{\mathtt{slhs} \leftarrow \mathit{false}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs = srhs_1\ op\ srhs_2;}\ \ \omega\rangle\phi,\ \Delta}$$

Description

- `op` is one of the JAVA comparison operators `<`, `<=`, `>`, and `>=`.
- `logOp` is the corresponding logical predicate to the JAVA operator `op`:
  - If `op` is `<`, then `logOp` is $<$.
  - If `op` is `<=`, then `logOp` is $\leq$.
  - If `op` is `>`, then `logOp` is $>$.
  - If `op` is `<=`, then `logOp` is $\geq$.

This rule symbolically executes a numeric comparison of two simple expression.

6.9 Rules for Handling Boolean Logical Operators

In this section, we give rules for handling boolean logical operators. Of course, these operators do not actually belong to integer arithmetic, but they occur very often in the context of arithmetical operations.

Rule 6.9.1 *binary-boolean-logical-operators*

Purpose Symbolic execution of expressions containing binary boolean logical operators.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\begin{array}{l}
1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\
2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_1),\ \Delta \\
3.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}_2),\ \Delta \\
4.\ \Gamma,\ \mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2 \vdash (\{\mathtt{slhs} \leftarrow \mathit{true}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\
5.\ \Gamma,\ \neg(\mathtt{srhs}_1\ \mathtt{logOp}\ \mathtt{srhs}_2) \vdash (\{\mathtt{slhs} \leftarrow \mathit{false}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\
\hline
\quad \Gamma \vdash \mu\langle \pi\ \mathtt{slhs}\ \mathtt{=}\ \mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2\mathtt{;}\ \ \omega\rangle\phi,\ \Delta
\end{array}$$

Description

- $\mathtt{op}$ is one of the boolean logical operators `&` and `|`.
- $\mathtt{logOp}$ is the corresponding logical connective for the logical JAVA operator $\mathtt{op}$:
  - If $\mathtt{op}$ is `&`, then $\mathtt{logOp}$ is $\wedge$.
  - If $\mathtt{op}$ is `|`, then $\mathtt{logOp}$ is $\vee$.

Rule 6.9.2 *unary-boolean-logical-operators*

Purpose Symbolic execution of expressions containing unary boolean logical operators.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\begin{array}{l}
1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\
2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\
3.\ \Gamma,\ \mathtt{srhs} \doteq \mathit{false} \vdash (\{\mathtt{slhs} \leftarrow \mathit{true}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\
4.\ \Gamma,\ \mathtt{srhs}_1 \doteq \mathit{true} \vdash (\{\mathtt{slhs} \leftarrow \mathit{false}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\
\hline
\quad \Gamma \vdash \mu\langle \pi\ \mathtt{slhs}\ \mathtt{=}\ \mathtt{op}\ \mathtt{srhs}\mathtt{;}\ \ \omega\rangle\phi,\ \Delta
\end{array}$$

Description • op is one of the boolean logical operators `^` and `!`.

Rule 6.9.3 *conditional-and-operator*

Purpose Transformation of an expression containing a conditional "and" operator.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \Gamma \vdash \langle \pi\ x\texttt{=true; if (!}srhs_1\texttt{)}\ x\texttt{=false; else}\ x\texttt{=}srhs_2\texttt{;}\ slhs\texttt{=}x\texttt{;}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle \pi\ slhs\ \texttt{=}\ srhs_1\ \texttt{\&\&}\ srhs_2;\quad \omega\rangle\phi,\ \Delta}$$

Description • x is a new variable of type `boolean`.

According to the JAVA language specification, the second argument of a conditional "and" operator is only evaluated if the first argument evaluates to `true`.

Rule 6.9.4 *conditional-or-operator*

Purpose Transformation of an expression containing a conditional "or" operator.

Remark The rule for the box operator $[\cdot]$ is analogous.

$$\frac{1.\ \Gamma \vdash \langle \pi\ x\texttt{=false; if (}srhs_1\texttt{)}\ x\texttt{=true; else}\ x\texttt{=}srhs_2\texttt{;}\ slhs\texttt{=}x\texttt{;}\ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle \pi\ slhs\ \texttt{=}\ srhs_1\ \texttt{||}\ srhs_2;\quad \omega\rangle\phi,\ \Delta}$$

Description • x is a new variable of type `boolean`.

According to the JAVA language specification, the second argument of a conditional "or" operator is only evaluated if the first argument evaluates to `false`.

6.10 Soundness and Completeness

6.10.1 Soundness

Soundness of the calculus $\mathcal{C}_{DLJ}$ (Corollary 6.10.2) is based on the following theorem, which states that all rules preserve the validity of the derived sequent.

Theorem 6.10.1
For all rule schemata of the calculus $\mathcal{C}_{DLJ}$, the following holds:

If all premisses of a rule schema instance are valid sequents for all possible implementations of method `overflow`*, then its conclusion is a valid sequent for all possible implementations of* `overflow` *as well.*

Corollary 6.10.2 (Soundness)
If a sequent $\Gamma \vdash \Delta$ is derivable with the calculus $\mathcal{C}_{DLJ}$, then it is valid in DLJ for all possible implementations of method `overflow`*, this means, $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is a valid formula for all implementations of* `overflow`*.*

Proving Theorem 6.10.1, the soundness of each rule has to be proved separately. Proving the soundness of program transformation rules, one has to show that the program contained in the premiss is equivalent to the program contained in the conclusion. For example, to prove the soundness of the following rule (Rule 6.3.4)

$$\frac{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}_2 \texttt{ = } \mathtt{expr};\ \mathtt{slhs}_1 \texttt{ = } \mathtt{slhs}_2;\ \ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle \pi\ \mathtt{slhs}_1 \texttt{ = } \mathtt{slhs}_2 \texttt{ = } \mathtt{expr};\ \ \omega\rangle\phi,\ \Delta}$$

one has to show that for all states s, t holds:

$$s\,[\![\mathtt{slhs}_2 \texttt{ = } \mathtt{expr};\ \mathtt{slhs}_1 \texttt{ = } \mathtt{slhs}_2;\ ]\!]\,t$$

if and only if

$$s\,[\![\mathtt{slhs}_1 \texttt{ = } \mathtt{slhs}_2 \texttt{ = } \mathtt{expr};\ ]\!]\,t.$$

All rules that symbolically execute a JAVA program construct are assignment rules, where the right-hand sides of the assignment differ. The soundness of an assignment rule for a simple while-language has been in proved in [Schlager, 2000].[4] The proof is based on a substitution lemma and is technically involved. Basically, the proofs of the assignment rules in this thesis would be similar. Of course, there are some more details (e. g. exceptions) that have to be taken into account because JAVA is a very complex programming language compared to a simple while-language.

6.10.2 Completeness

The calculus $\mathcal{C}_{DLJ}$ is *relatively* complete. This means, it is complete up to the handling of the domain of computation (the data structures). It is complete if the following "oracle rule" for arithmetic is available

[4] In [Schlager, 2000], the soundness of the assignment rule is not proved for the modality $\langle\cdot\rangle$ but for an modality "throughout", which states that a property has to be hold in each state of the execution of the program and thus, in particular, in the final state.

$$\frac{}{\Gamma \vdash \Delta}$$

where Γ, Δ are pure first-order formulas and $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is a valid arithmetical formula.

Of course, the first-order formulas that are valid in arithmetic are not even enumerable. Therefore, in practice, the oracle can only be approximated, and the above rule must be replaced by a rule (or set of rules) for computing respectively enumerating a *subset* of all valid first-order formulas (in particular, these rules must include equality handling). This is not harmful to "practical completeness". Rule sets for arithmetic are available, which—as experience shows—allow to derive all valid first-order formulas that occur during the verification of actual programs.

Typically, an approximation of the arithmetical "oracle" contains a rule for structural induction. An induction rule, however, is not only used to approximate the arithmetic "oracle" but is indispensable for completeness. It not only applies to first-order formulas but also to DLJ formulas containing programs.

Theorem 6.10.3
If a sequent is valid in $\mathcal{S}_{KeY}$ for all possible implementations of `overflow`*, then it is derivable with $\mathcal{C}_{DLJ}$.*

Corollary 6.10.4 (Completeness)
If ϕ is a valid DLJ formula, then the sequent $\vdash \phi$ is derivable with $\mathcal{C}_{DLJ}$.

7 Implementation

In this chapter we describe how the rules of the calculus presented in Chapter 6 are implemented and integrated into the KeY prover. At first, in Section 7.1, we introduce the language of *schematic theory specific rules* used for the implementation of the sequent calculus rules. Then, in Section 7.2, we show some of the rules of our calculus implemented as schematic rules.

7.1 Schematic Theory Specific Rules

For the implementation of the rules of the calculus we use the language presented in [Habermalz, 2000]. This language allows to express knowledge of specific theories in the form of proof rules which are called *schematic theory specific rules* (STSRs)[1]. In particular, a STSR contains pure logical knowledge, information, how this knowledge should be used, and information, when and where this knowledge should be presented for interactive application or applied automatically. A prototype implementation of a prover using STSRs is the interactive proof system IBIJa[2] [Supp, 1998].

Originally, the language of STSRs was designed for pure first-order predicate logic. Since the logical basis of the KeY project is DL, this requires some extensions of STSRs. In the following we will give an overview about STSRs (as they were originally defined) and some of the extensions necessary for DL concerning our work.

An STSR basically consists of three parts:

- the application part,
- the goal description, and
- the heuristic part.

In the following we give a short introduction into each part.

The *application part* has the form

[1] Since some time, in the KeY project, the name *taclet* is used instead of STSR.

[2] The acronym is derived from the German *Interaktiver Beweiser in Java* which can be translated as *Interactive Prover in Java*

`if` (*ifseq*) `find` (*findexp*) `varcond` (*varexplist*),

where *ifseq* is a sequent, *findexp* a term, formula, or a sequent with not more than one formula, and *varexplist* a list of elements of the form

x `new`,

`typeof(`x`)` y `new` (extension of [Habermalz, 2000]), and

y `not free in` z

where x, y, and z are meta variables which can stand for terms or variables. The `if`-part determines whether a STSR is applicable to a certain sequent. If there is no `if`-part, the STSR is applicable to any sequent. Within a sequent, the `find`-part (which may also be omitted) is used to determine the term of formula an STSR applies to. The `varcond`-part is used to describe certain properties of variables, for example, that a variable has to be "new", which is required often by the rules of our calculus. The extension of the `varcond`-part by the element "`typeof(`x`)` y `new`" offers the possibility to introduce a "new" variable y of the type of another variable x or even of the type of a JAVA expression. This is necessary for rules which transform a JAVA program into another program. To ensure that the original and the transformed program are semantically equivalent, it is often necessary to introduce a "new" variable of the type of a JAVA expression.

The application part can be seen as a description of the conclusion of a sequent calculus rule. On the other hand, the *goal description part* describes the premisses of a sequent calculus rule. It consists of a list of elements of the form

`replacewith` (*rwexp*) `add` (*addseq*) `addrules` (*stsrlist*)

and the single elements of the list are separated by semicolons. The term (formula) *rwexp* in the `replacewith`-part replaces the term (formula) *findexp* in the application part. The `add`-part allows to add additional sequents, for example if a sequent calculus rule has more than one premiss. A special feature of STSRs is the possibility to add new rules to the set of rules during a proof. The rules to add are specified in the `addrules`-part of a STSR.

The third main part of a STSR is the *heuristic part* and has the form

`heuristics` (*heunamelist*)

where *heunamelist* is a list of heuristic names.

Combining the three main parts, the general structure of an STSR is depicted in Figure 7.1.

```
name {
  if (ifseq) find (findexp)
    varcond u_1 new,..., u_m new,
    typeof(v_1) w_1 new,..., typeof(v_n) w_n new,
    x_1 not free in y_1 ,..., x_p not free in y_p
  replacewith (rwexp_1) add (addseq_1) addrules (r_{1,1},...,r_{1,ρ_1});
  ⋮
  replacewith (rwexp_q) add (addseq_q) addrules (r_{q,1},...,r_{q,ρ_q})
  heuristics (heu_1,...,heu_r)
};
```

Figure 7.1: This figure shows the general structure of a schematic theory specific rule (STSR).

7.1.1 Interactive and Non-Interactive Proving with STSRs

Most rule applications are triggered by the occurrence of certain terms and formulas specified in the application part of an STSR. By clicking on a term or formula in the user-friendly graphical user interface of the KeY prover, all possible rules that are applicable in this certain situation are offered. A typical situation of interactive rule application is shown in Figure 7.2. But to be practically useful, interactive proving must be enhanced with automatically performed proof steps as much as possible because even the proof of a small program requires a considerable number of rule applications. Automatic rule applications are performed by heuristics which can be separately turned on and off by the user. The set of STSRs that belong to a certain heuristic is defined in the heuristic part of an STSR. Note, that one STSR can belong to several heuristics.

7.2 The Rules of our Calculus as STSRs

Before we describe how the rules of the calculus in Chapter 6 are implemented as STSRs, we have to impose some restrictions concerning the implementation. Since the KeY prover is currently still under development, not all rules of the calculus can be implemented as STSRs.

The first restriction concerns the predicate $isdef(\boldsymbol{x})$ which is true if and only if the access to the value of the variable or attribute $\boldsymbol{x}$ is defined, that is the evaluation of $\boldsymbol{x}$ does not cause an exception. This definedness check cannot yet be performed by the prover at the moment and thus, premisses of the calculus containing the definedness predicate are omitted in the rules implemented as STSRs. If the prover

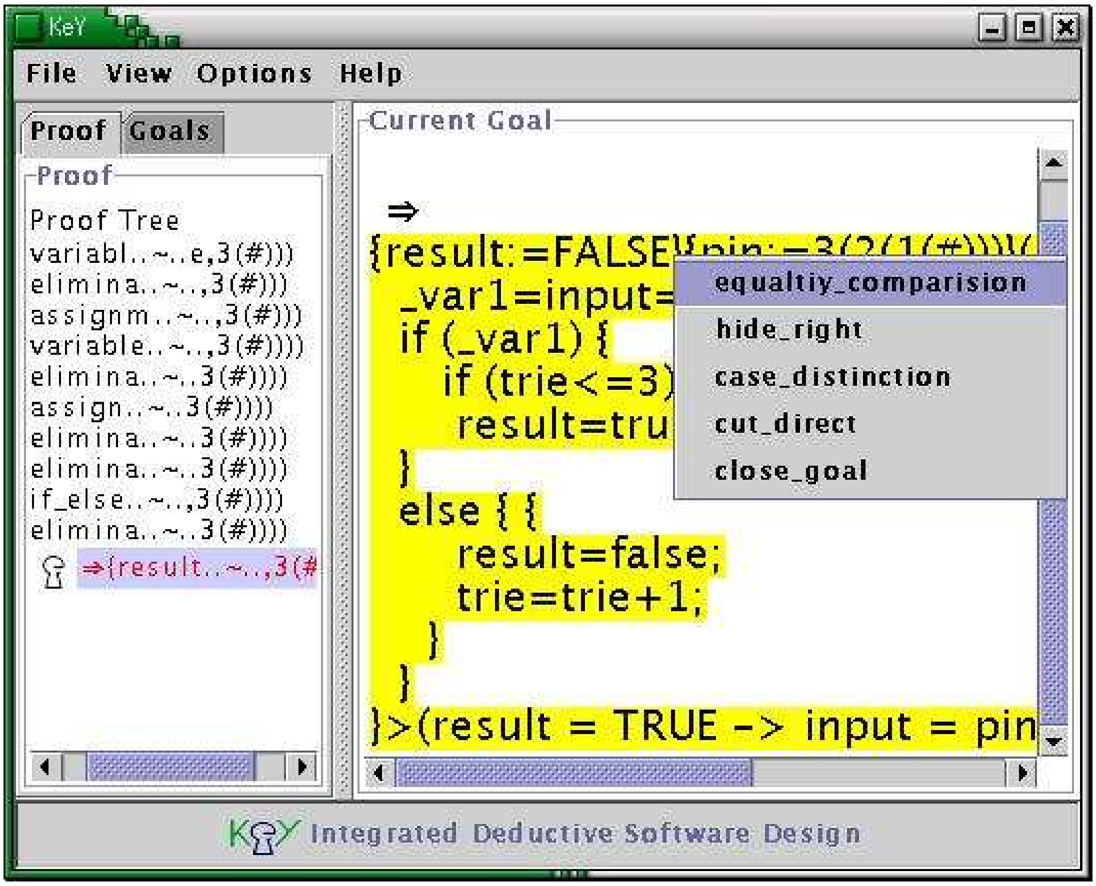


Figure 7.2: This figure shows a screen shot of an interactive rule application during proof construction with the KeY prover.

is capable of performing the definedness check, the STSRs do not totally have to be rewritten but one has only to add the missing premisses.

Another thing, that is currently not yet possible, is to simultaneously use the arithmetical types and the built-in types in one program. The reason for this restriction is a problem concerning the interface between the CASE tool TOGETHER CONTROL CENTER and the KeY prover. The solution of this problem goes beyond the scope of this thesis.

Before we present some examples of our rules implemented as STSRs, we first describe the structure of the files in which the STSRs are stored.

A rule file basically has the following structure.

`sorts`{*sortlist*}
`schema variables`{*svlist*}
`functions`{*funclist*}
`predicates`{*predlist*}
`heuristics`{*heulist*}
`rules`{*rulelist*}

The parts `sorts`, `schema variables`, `functions`, `predicates`, and `heuristics` contain the definition of the signature used in the `rules`-part which contains the STSRs. The only part that needs some explanation is the `schema variables`-part.

A schema variable is a special kind of variable that is used in STSRs, for example in the `find`- and `replacewith`-part to match a specific JAVA program construct or a specific term or formula. It is even possible to define schema variables that only match JAVA expressions of a certain type. If a STSR is applicable to a term or a formula, the meta variables in the STSR are instantiated with the matched expressions (JAVA expression, term or formula). The list *svlist* of schema variables declared in the `schema variables`-part has the form

$type_1$ `#`$u_1, \ldots,$`#`u_m;
$\vdots$
$type_n$ `#`$v_1, \ldots,$`#`v_p;

where $type_i$ is the type of the schema variable which defines what kind of expression is matched by the schema variable. The name of a schema variable is always preceded by the symbol "`#`".

In the following we describe the new types of schema variables introduced for the implementation of our rules as STSRs.

- `program simple lefthandside #slhs`

  A schema variable of this type corresponds to the meta variable ***slhs*** used in the rules of the sequent calculus.

- `program simple expression #se`

  A schema variable of this type corresponds to the meta variable ***srhs*** used in the rules of the sequent calculus.

- `program simple javabyte expression #seByte`

  A schema variable of this type corresponds to the meta variable ***srhs*** used in the rules of the sequent calculus with the restriction that the type of the matched expression has to be `byte`.

- `program simple javashort expression #seShort`

  Analogous to `program simple javabyte expression`.

- `program simple javaint expression #seInt`

  Analogous to `program simple javabyte expression`.

- `program simple javalong expression #seLong`

  Analogous to `program simple javabyte expression`.

- `program simple javabyteshortint expression #seByteShortInt`

  A schema variable of this type corresponds to the meta variable $\mathtt{srhs}$ used in the rules of the sequent calculus with the restriction that the type of the matched expression has to be `byte`, `short`, or `int`.

- `program nonsimple expression #nse`

  A schema variable of this type matches expressions described by the non-terminal symbol *nonSimpleExpr* in the grammar of integer arithmetic given in Section 6.1.

- `program expression #e`

  A schema variable of this type matches any JAVA expression and corresponds to the meta variable $\mathtt{expr}$ used in the rules of the sequent calculus rules.

- `program variable #v`

  Schema variables of this type are used in the `varcond` part of STSRs to introduce "new" program variables.

Now, that we have described the structure of STSRs and the schema variables used, we present some rules of the sequent calculus in Chapter 6 implemented as STSRs.

The first example is Rule 6.3.4 which transforms iterated assignment statements. For a better comparison, we repeat the definition of this rule:

$$\frac{1.\ \ \Gamma \vdash \mu\langle\pi\ \mathtt{slhs}_2 = \mathtt{expr};\ \ \mathtt{slhs}_1 = \mathtt{slhs}_2;\ \ \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}_1 = \mathtt{slhs}_2\ =\ \mathtt{expr};\ \ \omega\rangle\phi,\ \Delta}$$

Implemented as STSR, this rule looks like as follows:

```
iterated_assignments{
  find (<{..#slhs0=#slhs1=#e; ...}>post)
  replacewith (<{..#slhs1=#e; #slhs0=#slhs1; ...}>post)
  heuristics(simplify_expression)
};
```

As one can see, the premiss of the sequent rule corresponds to the `replacewith`-part of the STSR and the conclusion corresponds to the `find`-part. The fact, that the STSR in the `find`-part tries to "find" the conclusion of the sequent calculus rule shows that the rule is actually applied from bottom to top, as it is typical for proof construction in sequent calculi. The `heuristics`-part in this STSR defines that the rule belongs to the heuristic `simplify_expression`. Thus, this rule can be applied non-interactively if the heuristic `simplify_expression` is enabled.

A more complex rule which requires the introduction of a "new" variable of a certain type, is Rule 6.3.6 which, for the multiplication operator, is defined as follows.

$$\frac{1.\ \Gamma \vdash \mu\langle\pi\ \texttt{y = } \mathit{expr}_1\texttt{; z = } \mathit{expr}_2\texttt{;}\quad \mathit{slhs}\ \texttt{= y * z;}\quad \omega\rangle\phi,\ \Delta}{\Gamma \vdash \mu\langle\pi\ \mathit{slhs}\ \texttt{=}\ \mathit{expr}_1\ \texttt{*}\ \mathit{expr}_2\texttt{;}\quad \omega\rangle\phi,\ \Delta}$$

A requirement for the applicability of this rule is, that at least either of expr_1 and expr_2 is a non-simple expression. To express this requirement in the STSR, in fact three rules are necessary, namely one for the case, that both expressions are non-simple expressions and two for the cases that either expr_1 or expr_2 is a non-simple expression. The STSR we show applies to the case, that expr_1 is a non-simple expression and expr_2 is a simple expression.

```
compound_multiplication_simplification{
  find (<{..#slhs=#nse * #se;...}>post)
  varcond (typeof(#nse) #v new)
  replacewith (<{.. #typeof(#nse) #v=#nse; #slhs=#v * #se; ...}>post)
  heuristics (simplify_expression)
};
```

In the `varcond`-part, a "new" variable of the type of the expression matched by the schema variable `#nse` is introduced. This "new" variable has to be declared in the Java program contained in the `replacewith`-part. This is done by the `#typeof` construct which is instantiated with the type of `#nse` after applying this STSR. Note, that `typeof` in the `varcond`-part belongs to the language of STSRs, whereas `#typeof` in the `replacewith`-part is a meta construct similar to a schema variable.

As a final example, we regard Rule 6.4.2, handling binary operators applied to simple expressions of the arithmetical types. For the implementation we use a modified version of this rule, where the formula

$$\mu\langle\pi\ \mathit{slhs}\ \texttt{= overflow(}\mathit{srhs}_1\texttt{,}\mathit{srhs}_2\texttt{,"*");}\quad \omega\rangle\phi$$

is omitted and the premisses number five, six, and seven are replaced by one single premiss. The explanation why this is possible is given in the description of Rule 6.4.2.

Thus, for the multiplication operator, the rule looks like as follows.

$$\frac{\begin{array}{l}1.\ \Gamma \vdash \mathit{isdef}(\texttt{slhs}),\ \Delta \\ 2.\ \Gamma \vdash \mathit{isdef}(\texttt{srhs}_1),\ \Delta \\ 3.\ \Gamma \vdash \mathit{isdef}(\texttt{srhs}_2),\ \Delta \\ 4.\ \Gamma,\ \mathit{in}_{T_1}(\texttt{srhs}_1),\ \mathit{in}_{T_2}(\texttt{srhs}_2) \vdash \mathit{in}_T(\texttt{srhs}_1 * \texttt{srhs}_2),\ \Delta \\ 5.\ \Gamma \vdash (\{\texttt{slhs} \leftarrow \texttt{srhs}_1 * \texttt{srhs}_2\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma \vdash \mu\langle \pi\ \texttt{slhs = srhs}_1 \texttt{ * srhs}_2\texttt{;}\quad \omega\rangle\phi,\ \Delta}$$

The corresponding STSR is defined as follows.

```
assignment_multiplication{
  find (==> <{.. #slhs=#se0 * #se1; ...}>post)
  replacewith (==> {#slhs:=mul(#se0,#se1)}<{.. ... }>post);
  replacewith (#inType(#se0) & #inType(#se1) ==> #inType(#se0*#se1))
  heuristics (simplify_expression)
};
```

As mentioned in the beginning of this section, the first three premisses of the sequent calculus rule are omitted in the implementation as STSR since the KeY prover not yet supports this definedness check. In contrast to the first two examples, the `find`-part of this STSR tries to “find” a whole sequent instead of a formula (the arrow “`==>`” in the STSR corresponds to the sequent symbol “ $\vdash$ ”). The reason for that is the second `replacewith`-statement in the STSR which introduces an new sequent (which corresponds to the premiss number four in the sequent calculus rule). Since the types of the arguments of the `find`- and `replacewith`-part have to be equal, the argument of `find` has to be a sequent as well. The `#inType` construct in the STSR is a meta construct that generates the predicate in_T after the application of the STSR.

7.2.1 Rules for Handling Arithmetic on Literals

The application of a STSR, that symbolically executes a Java expression, translates integer literals contained in the expression into logical terms of the sort *number*. To handle arithmetical functions applied to terms of the sort *number*, the prototype implementation IBIJa contained a huge number of rules which, for example, calculate the sum of two numbers with the school method by adding digit by digit. For example, a typical rule to handle the addition of numbers in IBIJa, is the following which adds the last digits “3” and “1” of two numbers.

```
digit_31_add{
  find (+(3(i), 1(j)))
  replacewith (4(+(i, j)))
  heuristics (simplify_int)
};
```

Thus, the simplification of arithmetical functions applied to numbers takes a huge number of rule applications and a lot of time. Of course, the rules do not have to be applied by hand but can be applied non-interactively by a heuristic. Nevertheless, this approach is not suitable for us, since the number of necessary rules would too big. The rule file for integer arithmetic in IBIJa contains several hundred rules only to handle the addition, subtraction, and multiplication functions. Following this approach, we would by far need more than thousand rules for all the arithmetical operators (including the bit operators). Since there are already at lot of rules for handling the statements of the whole JAVA language, the overall number of rules would considerably slow down the prover.

Our approach for handling arithmetical functions applied to terms of the sort *number* is based on meta constructs. As mentioned before, a meta construct is similar to a meta variable and matches only certain terms or formulas which then are manipulated on the meta layer. Thus, for every arithmetical function we have a rule containing an according meta construct and the rule is applicable if and only if the arguments of the arithmetical function are of sort *number*. For example, the rule for the addition looks like as follows.

```
add_literals {
  find (+(i,j))
  replacewith (#add(i,j))
  heuristics (simplify_int)
};
```

After the application of this rule, the meta construct `#add(i,j)` is replaced by a term of the sort number which is the sum of `i` and `j`.

7.3 Simplification of Arithmetical Expressions

In this section we describe the rules for simplifying arithmetical expressions which we have integrated into the KeY prover. By arithmetical expressions we do not mean JAVA expressions but logical terms and formulas of the built-in integer types and the additional arithmetical types. By simplification rules we mean *rewrite rules* which substitute a term with a semantically equivalent and (mostly) less complex term.

Rules for simplifying arithmetical expressions are contained in all major proof systems like for example Isabelle [Paulson, 1994]. Since this system is popular and has been in use for many years, we adopt the simplification rules used in this system. In Isabelle, rules are combined to so called *theories* which may be built upon other theories. The theory, we are mainly interested in, is called *IntArith* and contains theorems about the integers which can be used to simplify expressions.

For the arithmetical types, we can adopt the simplification rules of Isabelle one-to-one (apart from the different syntax) but not all of them can by applied to terms of the built-in types because of their finiteness. For example, the formula

$$x + 1 < y + 1 \rightarrow x < y$$

which is a theorem over the integer numbers $\mathbb{Z}$, is not valid for the built-in types. A counter-example is to instantiate x with MAX_T and y with zero. Then, the result of MAX_$T + 1$ is MIN_T (which is less than zero) and thus, the premiss of the implication is true but the conclusion is false. On the other hand, the formula

$$x + 1 \doteq y + 1 \rightarrow x \doteq y$$

is a theorem as well for the arithmetical types as for the built-in types.

7.3.1 Simplification Rules for Terms of Arithmetical Types

As mentioned in the introduction, for the simplification of terms of the arithmetical types we can adopt the rules of Isabelle. To give an idea how these simplification rules look like, we present some examples.

The Isabelle theory *IntArith* contains the following theorem which expresses the distributivity of multiplication and addition.

```
theorem left_zadd_zmult_distrib:
i*u + (j*u + k) = (i+j)*u + k
```

The following rule is the according simplification rule in STSR notation.

```
left_zadd_zmult_distrib {
  find (+(mul(i,u),+(mul(j,u),k)) )
  replacewith (+(mul(+(i,j),u),k))
  heuristics (int_arithmetic)
};
```

Another Isabelle theorem expresses that the product of two integers m and n is equal to m if and only if either n is one or m is zero.

```
theorem zmult_eq_self_iff:
(m = m * n) = (n = 1 | m = 0)
```

As above, we also show the according rule as STSR.

```
zmult_eq_self_iff {
  find (i0=mul(i0,i1))
  replacewith (i0=0(#) | i1=1(#))
  heuristics (int_arithmetic)
};
```

Note, that in both examples, the Isabelle theorems are applicable as well from "left to right" as from "right to left" since the theorems are equations. In STSR notation, the equivalence of the left- and right-hand side is lost, since the `find`- and `replacewith`-part either contains the left- or the right-hand side of the equation. Thus, if one wants to have a rule that applies the equation from "right to left", another STSR is needed where the contents of the `find`- and `replacewith`-part are permuted.

7.3.2 Simplification Rules for Terms of Built-in Types

The built-in JAVA types have a finite range and the arithmetical operators acting on them cause overflow if the result is not representable in the corresponding type. Therefore, not all of the simplification rules for terms of the arithmetical types can be applied to terms of the built-in types. However, as shown in Definition 2.2.14, overflow can be expressed by the modulo operator and thus, all the theorems for modulo arithmetic can be applied to built-in types as well. In Isabelle, there exists the theory *IntDiv* which contains rules for the division and modulo operators.

The following STSR rewrites the modulo operator with its definition.

```
mod_rewr_base {
  find (mod(i,j) = r)
  varcond (k not free in i, k not free in j)
  replacewith ( ex k.(i=+(mul(k,j),r)))
};
```

The condition in the `varcond`-part guarantees that free occurrences of variables are not bound by the existential quantifier after the application of the rule.

A special case of the previous rule is the following, where the remainder of the modulo operation is zero.

```
mod_rewr {
  find (mod(i,j) = 0(#))
  varcond (k not free in i, k not free in j)
  replacewith (ex k.(i=mul(k,j)))
};
```

The following three Isabelle theorems, which express the distributivity of the modulo operator for multiplication, addition, and subtraction are very important for modulo arithmetic.

```
theorem zmod_zmult_eq:
(a * b) mod c = (a mod c) * (b mod c)

theorem zmod_zadd_eq:
(a + b) mod c = (a mod c) + (b mod c)
```

```
theorem zmod_zsub_eq:
(a - b) mod c = (a mod c) - (b mod c)
```

Note, that for the division operator the distributivity of the modulo operator does *not* hold. A counter example is $(4\ div\ 2)\ mod\ 2 = 0$, whereas $(4\ mod\ 2)\ div\ (2\ mod\ 2)$ is not defined (division by zero because $2\ mod\ 2 = 0$).

For the above theorems, we only present the STSR for the multiplication operator. The STSRs for the other theorems are analogous.

```
theorem zmod_zmult_eq {
  find (mod(mul(i0,i1),i))
  replacewith (mul(mod(i0,i),mod(i1,i)))
};
```

8 Example

In this chapter, we present an extended example that shows the usefulness of our approach.

8.1 PIN-Check Example

In the example we present, we describe the specification, implementation, and verification of a PIN-check module for a cash dispensers.

8.1.1 Natural Language Specification

Before we give an informal specification of the PIN-check module of a cash dispenser, we describe the scenario of a customer withdrawing money.

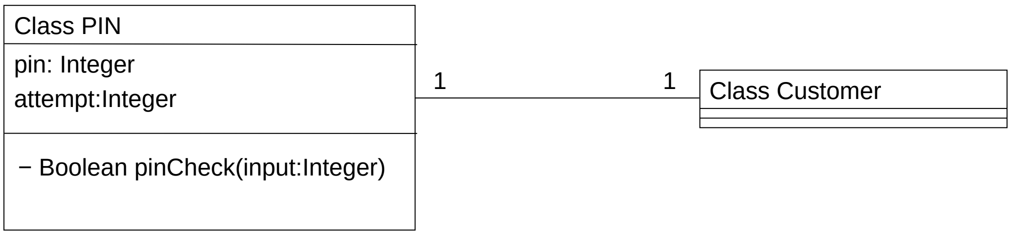
First, the customer inserts his credit card and then is prompted for his PIN. If the PIN was correct, the customer may withdraw money and then gets his credit card back. Otherwise, if the PIN was incorrect, the customer has another two attempts to enter the correct PIN. If the customer has entered an incorrect PIN more than two times, he is still able to enter another PINs but even if one of these PINs is correct, the customer cannot withdraw money and the credit card is retained to prevent misuse.

Our PIN-check module should contain a boolean method `pinCheck` that checks whether the PIN entered is correct and the number of attempts is less or equal three. Before we present a formal UML/OCL specification in the next section, we summarise the properties the method `pinCheck` must fulfil.

- The result value of `pinCheck` is `true` if the PIN entered is correct and if the number of attempts is less or equal three.
- The result value of `pinCheck` is `false` false if the PIN entered is incorrect or the number of attempts is greater than three.

8.1.2 Formal UML/OCL Specification

Now we present the formal UML/OCL specification of the PIN-check module which is depicted in Figure 8.1. The UML class diagram consists of the two classes `PIN`




```
context PIN::pinCheck(input:Integer):Boolean
post: result=true implies input=pin and attempt<=3
```

Figure 8.1: The UML/OCL specification of the PIN-check example.

and `Customer`. Instances of the class `PIN` represent the PIN of exactly one customer (instance of the class `Customer`) and one customer has exactly one PIN. This fact is expressed by the one-to-one association between `PIN` and `Customer`.

The actual PIN check is task of the method `pinCheck` in class `PIN` and the informal specification given in the previous section is now formalised as an OCL constraint. The OCL constraint is also depicted in Figure 8.1 and consists of a pre- and post-condition. The postcondition `result=true implies input=pin and attempt<3` specifies that the return value of the method `pinCheck` is true if and only if `input` (the PIN entered) is equal to `pin` (the correct PIN of the customer) and the number of attempts is less or equal three (this means the variable `attempt` is less than three since it is initialised with zero).

As one can see, this specification is incomplete and thus, does not correspond to the informal specification. In this simple example, it is quite easy to see that the formal specification is not adequate, but in more complex specifications it is not trivial to check that the formal specification really corresponds to the informal specification. Thus, the problems arising from an incomplete specification which are pointed out in the next section, may also occur in practice and not only in our little, admittedly somewhat constructed, example.

8.1.3 Implementation and Verification

Without our additional arithmetical types, a possible implementation could be the one depicted in Figure 8.2. In particular, this implementation might come from a programmer that does not take overflow into account.

This implementation of `pinCheck` basically consists of a non-terminating while-loop which can only be left with the statement `return true;`. In the body of the loop,

```
class PIN {

  private int attempt=0;
  private int pin=1234;

  public boolean pinCheck() {
    int input;
    while (true) {
      attempt++;
      input=promptForPIN();
      if (input==pin && attempt<=3)
        return true;
    }
  }

}
```

Figure 8.2: A possible implementation for the method `pinCheck` without using our additional arithmetical types.

at first the counter `attempt` is incremented by one and the method `promptForPin` is invoked, which returns the PIN entered by the user which then is assigned to the variable `input`. In case the entered PIN is equal to the actual PIN of the user and the number of attempts is less or equal three, the loop and thus the method is left with the statement `return true;`.

The generation of proof obligations from the specification and implementation yields the following DL formula, where the body of the loop is abbreviated with p.

$$\vdash \langle p \rangle \texttt{result} \doteq \mathit{true} \rightarrow \texttt{input} \doteq \texttt{pin} \wedge \texttt{attempt} <= 3$$

This sequent is derivable in our calculus. Thus, due to the correctness of the rules, the formula

$$\langle p \rangle \texttt{result} \doteq \mathit{true} \rightarrow \texttt{input} \doteq \texttt{pin} \wedge \texttt{attempt} <= 3$$

is valid in $\mathcal{S}_{KeY}$ and thus, in particular in $\mathcal{S}_{Java}$. Therefore, the above implementation is said to be correct, this means it satisfies the specification.

But this implementation has a behaviour that is probably not intended by the programmer. Suppose, the credit card was stolen and the thief wants to withdraw money but does not know the PIN. Thus, he has to try all possible PINs. Actually, according to the informal specification, after three wrong attempts any further attempt should not be successful any more. But if he does not give up, sometime the counter `attempt` will overflow and then has the negative value `MIN_int`. Then, the thief has many attempts to enter the correct PIN and thus, to withdraw money.

Of course, one reason for the unexpected behaviour of this implementation is the incomplete specification. But in the following we will demonstrate that this unintended behaviour of the program can be detected and thus be avoided following our approach.

Following our approach, in the implementation we would use the arithmetical type `arithInt` for the variable `attempt` instead of the built-in type `int`. This results in a proof obligation similar to the one above. The only difference is that the variable `attempt` in the body of the method is now of type `arithInt` instead of `int`. Trying to derive this proof obligation, after applying Rule 6.4.2 and some simplifications one gets the goal

$$\mathit{in}_{\texttt{arithInt}}(\texttt{attempt}),\ \mathit{in}_{\texttt{arithInt}}(1)\ \vdash\ \mathit{in}_{\texttt{arithInt}}(\texttt{attempt}+1).$$

But the above sequent is not derivable in our calculus. For the sequent to be valid and thus to be derivable, the formula

$$\mathit{in}_{\texttt{arithInt}}(\texttt{attempt}) \land \mathit{in}_{\texttt{arithInt}}(1) \to \mathit{in}_{\texttt{arithInt}}(\texttt{attempt}+1)$$

must hold in all states and in particular in states where `attempt` has the value `MAX_int`. But in such states the addition causes overflow and the above formula does not hold because the premiss is true and the conclusion is false (because $\texttt{attempt}+1$ is not in valid range).

Note, that this error has been uncovered by using our additional arithmetical types and our semantics $\mathcal{S}_{KeY}$. If the built-in types are used in the implementation, this error is not detected.

Since the proof obligation is not derivable in our calculus due to overflow, one has to correct the implementation to be able to prove its correctness. For example, one has to check whether the value of `attempt` is less than 3 before it is incremented. This results in the implementation depicted in Figure 8.3. To be absolutely sure that the implementation meets the informal specification, one actually also has has to prove that the class invariant $0 \leq \texttt{attempt} \leq 3$ holds.

The resulting proof obligation can now be derived in our calculus and thus, from Corollary 5.4.6 follows that no overflow occurs if the type `arithInt` is replaced with `int` in order to execute the program on the JAVA virtual machine. Thus, with this implementation, it cannot happen that a customer has more than three attempts to enter the valid PIN and withdraw money since no overflow occurs.

To conclude, as mentioned above, the main problem in this example is the inadequate specification (because it is incomplete) which is satisfied by the first implementation.

```
class PIN {

  arithInt attempt=0;
  private int pin=1234;

  public boolean pinCheck() {
    int input;
    while (true){
      if (attempt<3)
        attempt++;
      else
        attempt=4;
      input=promptForPIN();
      if (input==pin && attempt<=3)
        return true;
    }
  }

}
```

Figure 8.3: A possible implementation for the method `pinCheck` using the arithmetical type `arithInt`.

But due to unintended overflow, this implementation has a behaviour probably not supposed by the programmer. Following our approach, this unintended behaviour is uncovered and the program cannot be verified until this problem arising from overflow is solved.

As the example in this section shows, our approach can also contribute to detect errors in the specification and thus, if a program containing arithmetical types cannot be verified due to overflow, one should always verify whether the specification is adequate.

9 Conclusion and Future Work

All parts should go together without forcing.
You must remember that the parts
you are reassembling were disassembled by you.
Therefore, if you can't get them together again,
there must be a reason.
By all means, do not use a hammer.

— IBM MAINTENANCE MANUAL
1925

9.1 Conclusion

In this thesis, we have presented an approach for the handling of JAVA integer arithmetic in the verification of JAVA programs.

The approach is based on an extension of the JAVA syntax by additional primitive types (called arithmetical types) and a semantics $\mathcal{S}_{KeY}$ for JAVA integer arithmetic. The introduction of the additional arithmetical types aims at preventing errors arising from unintended overflow which are often hard to find.

In Chapter 6, a sequent calculus for JAVA integer arithmetic, based on the extended syntax and semantics, is presented. The rules of this calculus have been implemented and integrated into the KeY prover. Additionally, we have enhanced the prover by rules for simplifying arithmetical expressions which are described in Section 7.3.

9.2 Future Work

As mentioned in Section 7.2, due to problems concerning the interface between TOGETHER CONTROL CENTER and the KeY prover, it is not yet possible to use the built-in types and the arithmetical types in one program simultaneously. Future work certainly includes the solution of this problem.

Another topic for future work would be an extension of the OCL by additional data types. The only integer type so far contained in the OCL is INTEGER which has a infinite range. But since all data types in programming languages are finite, this leads to problems which could be avoided if the OCL is extended by several finite data types with different ranges. Additionally, one could define that arithmetical operations on these finite types may cause overflow similar to the built-in Java types and thus, the problems arising from overflow have already to be tackled in the specification. This would help to prevent implementation errors arising from overflow.

Bibliography

[Ahrendt et al., 2000] Ahrendt, W., Baar, T., Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Menzel, W., and Schmitt, P. H. (2000). The KeY approach: Integrating object oriented design and formal verification. In Ojeda-Aciego, M., Guzman, I. P. d., Brewka, G., and Pereira, L. M., editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer.

[Beckert, 2001] Beckert, B. (2001). A dynamic logic for the formal verification of Java Card programs. In Attali, I. and Jensen, T., editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer.

[Beckert and Schlager, 2001] Beckert, B. and Schlager, S. (2001). A sequent calculus for first-order dynamic logic with trace modalities. In Gorè, R., Leitsch, A., and Nipkow, T., editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer.

[Börger and Schulte, 1999] Börger, E. and Schulte, W. (1999). A programmer friendly modular definition of the semantics of Java. In Alves-Foss, J., editor, *Formal Syntax and Semantics of Java*, LNCS 1523, pages 353–404. Springer.

[Bracha et al., 2000] Bracha, G., Gosling, J., Joy, B., and Steele, G. (2000). *The Java Language Specification, Second Edition*. Addison Wesley.

[Brien and Nicholls, 1992] Brien, S. and Nicholls, J. (1992). Z base standard. Technical Monograph PRG-107, Oxford University Computing Laboratory.

[Habermalz, 2000] Habermalz, E. (2000). Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Department of Computer Science, University of Karlsruhe.

[Harel, 1979] Harel, D. (1979). *First-order Dynamic Logic*. LNCS 68. Springer.

[Huisman, 2001] Huisman, M. (2001). *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, The Netherlands.

[Keller, 2002] Keller, U. (2002). Übersetzung von OCL Constraints in Formeln einer Dynamischen Logik für Java Card. Master's thesis, Universität Karlsruhe. Diplomarbeit, in german.

[OMG, 1999a] OMG (1999a). Object constraint language specification, version 1.3. chapter 7 in [OMG, 1999b]. OMG Document ad970808.

[OMG, 1999b] OMG (1999b). OMG unified modeling language spezification, version 1.3. OMG Document.

[Owre et al., 1992] Owre, S., Rushby, J., and Shankar, N. (1992). PVS: A protoype verification system. In Kapur, D., editor, *Proceedings, 11th International Conference on Automated Deduction (CADE), Saratoga Springs/NY, USA*, LNCS 607, pages 748–752. Springer.

[Paulson, 1994] Paulson, L. C. (1994). *Isabelle: A Generic Theorem Prover*. Springer, Berlin.

[Reif, 1995] Reif, W. (1995). The KIV-approach to software verification. In Broy, M. and Jähnichen, S., editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, volume 1009 of *LNCS*. Springer.

[Schlager, 2000] Schlager, S. (2000). Erweiterung der Dynamischen Logik um temporallogische Operatoren. Studienarbeit, Fakultät für Informatik, Universität Karlsruhe. In German. Available at: `http://i12www.ira.uka.de/~schlager/publications/Studienarbeit.ps.gz`.

[Stenzel, 2001] Stenzel, K. (2001). Verification of JavaCard Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany. Available at http://www.Informatik.Uni-Augsburg.DE/swt/fmg/papers/.

[Supp, 1998] Supp, M. (1998). Implementierung eines integriert interaktiven und reflexiven Beweisers für prädikatenlogik. Master's thesis, Universität Karlsruhe. Diplomarbeit, in german.

[von Oheimb, 2001] von Oheimb, D. (2001). *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Univerität München.