# Integer Arithmetic in the Specification and Verification of Java Programs

Bernhard Beckert and Steffen Schlager

University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
`beckert@ira.uka.de`, `schlager@ira.uka.de`

**Abstract.** In this paper we present an approach for handling integer arithmetic in the specification and verification of JAVA programs. In particular, problems like overflow and underflow arising from the finiteness of the JAVA types are tackled.

## 1 Introduction

*Background.* The work reported here has been carried out as part of the KeY project [1, 2] (see `i12www.ira.uka.de/~key`). The goal of KeY is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

– The specification language should be usable by people who do not have years of training in formal methods (we decided to use UML/OCL).
– The programs that are verified should be written in a "real" object-oriented programming language. We decided to use JAVA (actually KeY only supports the subset JAVA CARD, but the difference is not relevant for the topic of this paper).

The Unified Modeling Language (UML) [8] has been widely accepted as the standard object-oriented modeling language and is supported by a great number of CASE tools. The Object Constraint Language (OCL) is an integral part of the UML, and was introduced to express subtleties and nuances of meaning that diagrams cannot convey by themselves.

*The Problem.* For the verification component in the KeY system, we use a Dynamic Logic (which can be seen as a variant of Hoare Logic). This instance of Dynamic Logic, called JavaDL, can be used to reason about Java Card programs [3]. Obviously, the semantics of JavaDL arithmetical operations should on the one hand correspond to the behaviour of arithmetical operations in UML/OCL, and on the other hand must reflect the Java semantics. Unfortunately, it is hard to meet both requirements at the same time.

For specifying propositions about integers, UML/OCL provides the basic data type INTEGER. This data type, by definition, is isomorphic to the mathematical set $\mathbb{Z}$ of integers, and thus has an infinite range. The usual operators for addition, multiplication, and so on are available on INTEGER.

On the other hand, in JAVA there are the four primitive integer data types `byte`, `short`, `int`, and `long`, which all have a different but finite range. Moreover, the semantics of the arithmetical operators differs from the (usual) semantics they have in UML/OCL (the main reason for this is of course the finiteness of the data types). In Java, if the result of an arithmetical operation exceeds the range of its type, an *overflow* occurs, i.e., the result is computed modulo the size of the date type. For example, $\texttt{MAX\_int} + 1 = \texttt{MIN\_int}$.

*Our Solution.* In this paper, we present the approach for handling JAVA integer arithmetic used in the KeY project and has been implemented in the KeY system (a much more detailed account including proofs and a sequent calculus can be found in [9]). This approach is based on combining the integer semantics of UML/OCL and the integer semantics defined in the JAVA language specification [5]. Both semantics by themselves are not suitable: Using the semantics of UML/OCL may result in incorrect programs being verified, and using the semantics of JAVA may result in programs being verified that are merely "incidentally" correct and whose behaviour does not reflect the intentions of the programmer.

## 2 Two Different Semantics

### 2.1 From UML/OCL to DL Proof Obligations

To prove the correctness of a program, one has to prove the validity of Dynamic Logic formulas (*proof obligations*) which are generated from the specification and implementation. The approach for generating proof obligations used in the KeY project is described in [7,4].

Dynamic Logic can be seen as a modal predicate logic with a modality $\langle p \rangle$ for every program $p$ (we allow $p$ to be any sequence of legal JAVA statements); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program $p$. In standard DL there can be several of these states (worlds) because the programs can be non-deterministic; but here, since JAVA programs are deterministic, there is exactly one such world (if $p$ terminates) or there is no such world (if $p$ does not terminate). The formula $\langle p \rangle\phi$ expresses that the program $p$ terminates in a state in which $\phi$ holds. A formula $\phi \rightarrow \langle p \rangle\psi$ is valid if for every state $s$ satisfying the pre-condition $\phi$, a run of the program $p$ starting in $s$ terminates, and in the terminating state the post-condition $\psi$ holds.

### 2.2 UML/OCL Semantics $\mathcal{S}_{OCL}$

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 JAVA types `byte`, `short`, `int`, and `long` is interpreted in the same way as the OCL type INTEGER. Also, 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 infinite range and overflow is totally disregarded. Thus, it is easy to see that $\mathcal{S}_{OCL}$ differs from the semantics implemented by the virtual machine. Nevertheless, this semantics is used in many publications (e. g. [6, 11]) on the semantics of JAVA.

### 2.3 Java Semantics $\mathcal{S}_{Java}$

The second semantics, which is called $\mathcal{S}_{Java}$, exactly corresponds to the semantics defined in the JAVA language specification [5] (and thus to the semantics implemented by the virtual machine). Especially, this means that the primitive 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 [10]. 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 since the DL semantics and the virtual machine semantics are equivalent. 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 $\mathcal{S}_{Java}$ is the best choice. But there are also some drawbacks which are 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`[1] 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 stated using the type INTEGER with infinite range. If one is not aware of that fact during the specification and implementation, this might lead to an program that is merely "incidentally" correct. This means that a program fulfills its specification although overflow may occur, but the fact that overflow occurs was not intended neither by the modeler nor the programmer. Thus, the behaviour of the program during the execution may differ from the intended behaviour. For example, using $\mathcal{S}_{Java}$, the formula $\mathtt{i} > 0 \rightarrow \langle \mathtt{i=i+1;\ i=i-1} \rangle \mathtt{i} > 0$ is valid although in case the value of `i` is `MAX_int`, an overflow occurs and the value of `i` is (surprisingly) negative in the intermediate state after the first assignment. This phenomenon mostly occurs if the specification is incomplete or consists only of some safety constraints (like $\mathtt{i} > 0$). 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 (see the extended example in Section 4).

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

Of course, the above problem does not arise 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`.

## 3 A Combination of Both Worlds

### 3.1 The Basic Idea

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 $\forall x \exists y (y > x)$ are valid in our JavaDL as well. To achieve that, we extend the syntax of JAVA by the additional primitive 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 first obvious difference between semantics $\mathcal{S}_{KeY}$ and $\mathcal{S}_{OCL}$ or $\mathcal{S}_{Java}$ is, that the signatures of the underlying programming languages 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.

In $\mathcal{S}_{KeY}$, the additional arithmetical types basically have an infinite range. The operators acting on them have the same semantics as in $\mathcal{S}_{OCL}$ 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 implicitly defined method `overflow(x,y,op)`[2] whose behaviour remains unspecified. This means in case of overflow, the result is unspecified and the execution of the method `overflow` does not have to terminate.

Leaving `overflow` unspecified has two main advantages.

The first one is, that there is no reasonable implementation for the method `overflow`. In principle, there are three possible implementations but each of them is not suitable as we shortly explain 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 $\mathcal{S}_{Java'}$. As the name $\mathcal{S}_{Java'}$ suggests, it is similar to $\mathcal{S}_{Java}$ (the differences are discussed later). The problem of $\mathcal{S}_{Java}$, that programs are only "incidentally" correct, also applies to $\mathcal{S}_{Java'}$. An advantage of semantics $\mathcal{S}_{Java'}$ compared to $\mathcal{S}_{Java}$ is that, using the arithmetical types, formulas like $\forall x \exists y (y > x)$ are valid.
- The implementation of `overflow` is such that it never terminates. This implementation is not acceptable, because in this case, a formula like [`arithInt i=MAX_int+1;`]`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 possible implementations leads to a semantics that is not suitable for program verification.

The second advantage of leaving `overflow` unspecified concerns the connection between the derivability of a formula in our calculus and the behaviour of a program contained in this formula during the execution on the JAVA virtual machine.

[2] The parameter `op` is the operator that caused overflow and `x,y` are its arguments.

If a JavaDL formula $\phi$ is derivable in our calculus based on $\mathcal{S}_{KeY}$ (i. e. `overflow` is unspecified), then $\phi$ is valid in $\mathcal{S}_{KeY}$ for all implementations of `overflow` (this follows from the soundness of the calculus). In particular, $\phi$ is a valid formula for the second implementation of `overflow` mentioned above. Thus, $\phi$ is valid in semantics $\mathcal{S}_{Java'}$ 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 $\phi$ in semantics $\mathcal{S}_{KeY}$. If one only knows that $\phi$ is valid in semantics $\mathcal{S}_{Java'}$, this conclusion is not correct. However, the information, that no overflow occurs, is very important and is used in Theorem 2.

### 3.2 Properties of the Combined Semantics $\mathcal{S}_{KeY}$

The additional types of the extended JAVA have an infinite range, and thus, in a DL based on $\mathcal{S}_{Java'}$, there are states that do neither exist in DL based on $\mathcal{S}_{Java}$ nor in the virtual machine. In the following, we call such states "unreal" states[3], whereas states, that exist in both DLs, are called "real" states. But we are interested in the behaviour of a program during the execution on the 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, $\mathcal{S}_{Java}$ and $\mathcal{S}_{Java'}$ are equivalent (in both, the result of overflow is the same). In particular, this means, that in real states the semantics $\mathcal{S}_{Java}$ of the built-in types corresponds to the semantics $\mathcal{S}_{Java'}$ of the arithmetical types. Thus, in real states a program $p$ is equivalent to a program $\mathit{ptransf}(p)$, where the arithmetical types are replaced with the corresponding built-in types (see Theorem 3).

Corollary 1 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 $\mathit{ptransf}(p)$ on the JAVA virtual machine and after the execution, the property $\psi$ holds.

Note, that Corollary 1 does not apply to arbitrary formulas because a formula like $[p]\mathit{true}$ 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 a built-in 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 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 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. For example, the formula $\exists \mathtt{i}(\mathtt{i} > \mathtt{MAX\_}T \land \langle\mathtt{j=i+1;}\rangle \mathtt{j} \doteq \mathtt{i} + 1)$ is valid in $\mathcal{S}_{KeY}$ if `i,j` are of an arithmetical type $T$.[4] In this case, method `overflow` is not invoked during the symbolic execution in JavaDL, since the state, in which the addition is evaluated, is an unreal state. In this case, propositions about the behaviour of the transformed program during the execution on the virtual machine make no sense because unreal states cannot be reached by the virtual machine.

In Table 1, the properties of the three semantics $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, and $\mathcal{S}_{KeY}$ are summarised and compared.

Of course, 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

[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.

[4] Actually, a program variable must not be quantified and a formula like $\forall \mathtt{i}\phi(\mathtt{i})$ is a short form for the syntactically correct formula $\forall x \langle \mathtt{i=}x\mathtt{;}\rangle \phi(\mathtt{i})$.

| **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 1.** Comparison of some properties of $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, and $\mathcal{S}_{KeY}$.

built-in types before the program can be compiled and executed. We therefore define a transformation *ptransf* that replaces all arithmetical types in a program by the corresponding primitive JAVA types.

A consequence of applying the type transformation *ptransf* to a program $p$ before compiling and running it on the virtual machine is, that the actually executed program *ptransf*($p$) and the verified program $p$ differ. However, the following theorems show that this difference is harmless and does not affect the verified behaviour of $p$.

**Theorem 1.** *If a* JAVA *program* $p$ *is well-typed, then the program* $ptransf(p)$ *is well-typed.*

**Theorem 2.** *If* $\vdash_{KeY} \phi$ *(and, therefore,* $\models_{\mathcal{S}_{KeY}} \phi$*), then both* $\models_{\mathcal{S}_{OCL}} \phi$ *and* $\models_{\mathcal{S}_{Java'}} \phi$*.*

**Definition 1.** *Let* $s$ *be a real JavaDL state. The* isomorphic state $iso(s)$ *to* $s$ *is the virtual machine state, in which all state elements (program variables and fields) of arithmetical type in* $s$ *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. In the following theorem, $s\,[\![p]\!]_{\mathcal{S}_{Java'}}\,s'$ means that program $p$, started in state $s$, terminates in state $s'$ using the DL semantics $\mathcal{S}_{Java'}$.

**Theorem 3.** *Let* $p$ *be a* JAVA *program that may contain arithmetical types. Then, for all real states* $s$ *and all (arbitrary) states* $s'$*: If* $s\,[\![p]\!]_{\mathcal{S}_{Java'}}\,s'$*, then* $iso(s)\,[\![ptransf(p)]\!]_{\mathcal{S}_{Java}}\,iso(s')$*.*

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 $\Gamma$ holds in $s$ for all possible implementations of `overflow`.

**Corollary 1.** *Let* $\Gamma, \psi$ *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 JavaDL state.*

*If (i)* $\vdash_{KeY} \Gamma \to \langle p\rangle\psi$*, (ii)* $s \models_{\mathcal{S}_{KeY}} \Gamma$*, and (iii)* $s$ *is a real state, then, when the transformed program* $ptransf(p)$ *is started in* $iso(s)$ *on the virtual machine, (a) no overflow occurs and (b) the execution terminates in the state* $iso(s')$*.*

Some sample formulas and the necessary type to make the according formula valid in the different semantics are depicted in Table 2.

Below, 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 be avoided by using the additional arithmetical types.
- Formulas like $\forall x \exists y (y > x)$ that are intuitively valid over the integer numbers $\mathbb{Z}$, are still valid if $x, y$ are of an arithmetical type.

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

**Table 2.** This table shows some formulas and in the last three columns is denoted of which type the program variables `i` and `j` have to be that the according formula is valid in semantics $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, or $\mathcal{S}_{KeY}$.

### 3.3 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 is unspecified. By giving axioms, that `overflow` must satisfy, it is possible to define variants of $\mathcal{S}_{KeY}$.

For example, one could define, that `overflow` always terminates or is symmetric. If `overflow` always terminates, a formula like $\Gamma \rightarrow \langle p \rangle true$ is possibly derivable, even if overflow occurs during the execution of $p$ and `overflow` is invoked, since goals of the form $\Gamma \vdash \langle \mathtt{x=overflow(arg1,arg2,op);} \rangle true$ can 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 $\mathcal{S}_{Java'}$ of $\mathcal{S}_{KeY}$, Theorem 2 and Theorem 3 still hold.

### 3.4 Steps in Software Development

Following our approach, the steps in software development are the following.

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 our calculus, one has to derive the proof obligations generated from the specification and implementation using the translation described in [7, 4]. If all proof obligations are derivable, from Corollary 1 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 virtual machine and in particular, no overflow occurs.

## 4 Extended Example

In this example we describe the specification, implementation, and verification of a PIN-check module for a cash dispensers. Before we give an informal specification of the PIN-check module of the cash dispenser, we describe the scenario of a customer trying to withdraw 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 he 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, he 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. The informal specification of this method is, that the result value is `true` if and only if the PIN entered is correct and if the number of attempts is less or equal three.

The formal specification of the method `pinCheck` consists of the OCL postcondition

```
context PIN::pinCheck(input:Integer):Boolean
post: result=true implies input=pin and attempt<=3
```

```
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;
    }
  }
}
```

```
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;
    }
  }
}
```

**Fig. 1.** Implementation of method `pinCheck` without (left) and with (right) using the additional arithmetical type `arithInt`.

stating that the return value of `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.

In this simple example, it is easy to see that the formal specification is not adequate (this means does not correspond to the described scenario), 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 this example, may also occur in practice and not only in our simple example.

Without our additional arithmetical types, a possible implementation could be the one shown on the left in Figure 1. In particular, such an implementation may bw written by a programmer who 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, 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 user's correct PIN and the number of attempts is less or equal three, the loop and thus the method terminates with `return true;`.

The generation of proof obligations from the specification and implementation yields the following JavaDL formula, where the body of the loop is abbreviated with $p$:

$$\vdash \ \langle p\rangle \mathtt{result} \doteq \mathit{true} \rightarrow \mathtt{input} \doteq \mathtt{pin} \land \mathtt{attempt} <= 3$$

This sequent is derivable in our calculus. Therefore, due to the correctness of the rules, it is valid in $\mathcal{S}_{KeY}$ and, thus, in particular in $\mathcal{S}_{Java}$. Consequently, the above implementation is said to be correct, which means that 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`. Since nothing is known about `overflow`, the only way to derive this in our JavaDL calculus is to prove—as a lemma or sub-goal—that no overflow occurs (and, thus, `overflow` is not invoked). Therefore, after several proof steps and simplifications, one gets the goal

$$in_{\texttt{arithInt}}(\texttt{attempt}),\ in_{\texttt{arithInt}}(1)\ \vdash\ in_{\texttt{arithInt}}(\texttt{attempt}+1).$$

But the above sequent is neither valid nor derivable, because it is not true in states where `attempt` has the value `MAX_int`. 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 on the right side in Figure 1. To be absolutely sure that the implementation meets the informal specification, one actually also 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 1 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, the main problem in this example is the inadequate specification (because it is incomplete) which is satisfied by the first implementation. 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 check whether the specification is adequate.

## References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
2. W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering. 5th International Conference, FASE 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.
3. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
4. B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002. To appear. Available at `i12www.ira.uka.de/~key/doc/2002/BeckertKellerSchmitt02.ps.gz`.
5. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.
6. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.
7. U. Keller. Übersetzung von OCL Constraints in Formeln einer Dynamischen Logik für Java Card. Master's thesis, Universität Karlsruhe, 2002. Diplomarbeit, in german.
8. Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.
9. S. Schlager. Behandlung von Integer Arithmetik bei der Verifikation von Java-Programmen. Master's thesis, Universität Karlsruhe, 2002. Available at `i12www.ira.uka.de/~key/doc/2002/DA-Schlager.ps.gz`.
10. K. Stenzel. Verification of JavaCard Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available at `www.informatik.uni-augsburg.de/swt/fmg/papers/`.
11. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.