Software Verification with Integrated Data Type Refinement for Integer Arithmetic

Bernhard Beckert[1] and Steffen Schlager[2]

[1] University of Koblenz-Landau, Institute for Computer Science
D-56072 Koblenz, Germany
beckert@uni-koblenz.de

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

Abstract. We present an approach to integrating the refinement relation between infinite integer types (used in specification languages) and finite integer types (used in programming languages) into software verification calculi. Since integer types in programming languages have finite ranges, in general they are not a correct data refinement of the mathematical integers usually used in specification languages. Ensuring the correctness of such a refinement requires generating and verifying additional proof obligations. We tackle this problem considering Java and UML/OCL as example. We present a sequent calculus for Java integer arithmetic with integrated generation of refinement proof obligations. Thus, there is no explicit refinement relation, such that the arising complications remain (as far as possible) hidden from the user. Our approach has been implemented as part of the KeY system.

Keywords: Software verification, specification, UML/OCL, data refinement, Java, integer arithmetic.

1 Introduction

The Problem. Almost all specification languages offer infinite data types, which are not available in programming languages. In particular this holds for the mathematical *integer* data type which we will focus on in this paper. Infiniteness of integer types on the specification level is an important feature of a specification language for two reasons:[1]

1. Specifications should be abstract and independent of a concrete implementation language.
2. Developers think in terms of arithmetic on integers of unrestricted size.

[1] For these reasons, Chalin [7] proposes to extend the Java Modeling Language (JML) [18], which does not support infinite integer types, with a type `infint` with infinite range.

E. Boiten, J. Derrick, G. Smith (Eds.): IFM 2004, LNCS 2999, pp. 207–226, 2004.

In the implementation, the infinite types have to be replaced with finite data types offered by the programming language. Verifying the correctness of the implementation requires among other things to show that this replacement does not cause problems. Speaking in terms of refinement one has to prove that the finite types are a correct *data refinement* of the specification language types (in the particular context where they are used). This is done by generating additional proof obligations for each arithmetical expression stating that the result does not exceed the finite range of the type of the expression. By verifying these additional proof obligations, we establish that the programming language types are only used to the extent that they indeed are a refinement of the specification language types. This check cannot be done once and for all but has to be repeated for each particular program. It is tedious and error-prone if done by hand.

Our Solution. Our solution to the integer data refinement problem is to define a verification calculus that combines the infinite integer semantics of specification languages and the finite integer semantics of programming languages. To avoid "incidentally" correct programs (as defined below), we verify that no overflow[2] occurs during the execution of a program, i.e., a pre-condition is added to each arithmetical operation stating that its result is within the bounds of the Java data type. That is, we are not content with merely showing that a program satisfies its specification, which it may do even if an overflow occurs.

To keep all these complications hidden from the user as far as possible, the relation between the different types of integer semantics is not made explicit (there is no formal refinement relation). Instead, the handing of the refinement relation and, in particular, the generation of proof obligations to make sure that no overflow occurs, is integrated into the rules of the verification calculus.

Our Choice of Specification and Implementation Language. In this paper, we use Java as implementation language, and the specification language we consider is UML/OCL.

Note, however, that our particular choice of specification and implementation languages is not crucial to our approach. The languages UML/OCL and Java can be substituted by almost any other specification and implementation languages (e.g. Z [22] or B [1] resp. C++). We use UML/OCL and Java mainly because the work presented here has been carried out as part of the KeY project [2,3] (see `http://www.key-project.org`). 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. We decided to use UML/OCL as specification language since the Unified Modeling Language (UML) [19] has been widely accepted as

[2] The situation that the result of an arithmetical operation exceeds the maximum or minimum value of its type is called *overflow*. In Java, if overflow occurs the result is computed modulo the size of the data type. For example, $\texttt{MAX\_int} + 1 \doteq \texttt{MIN\_int}$ (where `MAX_int` and `MIN_int` are the maximum resp. minimum value representable in type `int`).

the standard object-oriented modelling language and is supported by a great number of CASE tools. 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).

Motivation for Our Solution. The motivation for our solution is that using the semantics of JAVA (as implemented by a JAVA Virtual Machine) to verify that a program correctly implements its specification (without checking for overflow) may still lead to undesired results if the specification is too weak. A formally correct program may not reflect the intentions of the programmer if overflow occurs during its execution—even if its observable behaviour satisfies the specification. Such programs, which we call "incidentally" correct, are a source of error in the software development process (as explained in Section 2.2). The problem is aggravated by the fact that JAVA, as well as many other programming languages like C++ and Pascal, do not indicate overflow in any way (in some other languages, such as Ada, an exception is thrown). Moreover, many JAVA programmers are not aware of this behaviour of JAVA integers.[3] But even programmers who know about this JAVA feature make errors related to overflow. For example, in [6] a flaw arising from unintended overflow in the implementation of Gemplus' electronic purse case study [17] is discovered. The result of this flaw is that the method `round` which is supposed to return the closest integer, in fact returns -32768 when invoked with 32767.999.

Dynamic Logic. For the verification component in the KeY system, we use an instance of Dynamic Logic. This instance, called JavaDL, can be used to specify and reason about properties of JAVA CARD programs [4].

Dynamic Logic (DL) is 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 without threads are deterministic (so far concurrency is not considered in the KeY system), 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 ϕ holds. A formula $\psi \to \langle p \rangle \phi$ is valid if for every state s satisfying the pre-condition ψ, a run of the program p starting in s terminates, and in the terminating state the post-condition ϕ holds.

To prove the correctness of a program, one has to prove the validity of DL formulas (*proof obligations*) that are generated from the UML/OCL specification and the JAVA implementation. The approach for generating proof obligations used in the KeY project is described in [5].

[3] The claim that many programmes are not aware of the behaviour of JAVA integers in case of an overflow is based on the authors' personal experiences made in teaching courses for computer science students and conversations with programmers working in industry.

Deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of Java.

Related Work. So far, research in this area mainly focused on formalising and verifying properties of floating-point arithmetic [12,13] (following the IEEE 754 standard). However, there are good reasons not to neglect integer arithmetic and in particular integer arithmetic on finite programming language data types. For example, integer overflow was involved in the notorious Ariane 501 rocket self-destruction, which resulted from converting a 64-bit floating-point number into a 16-bit signed integer. To avoid such accidents in the future the ESA inquiry report [9] explicitly recommended to "verify the range of values taken by any internal or communication variables in the software."

Approaches to the verification of Java programs that take the finiteness of Java's integer types into consideration—but not their relationship to the infinite integer types in specification languages—have been presented in [16,23].

The verification techniques described in [20,25,15] treat Java's integer types as if they were infinite, i.e., the overflow problem is ignored.

In [10], a problem in the Java Card language specification is pointed out. Certain Java Card programs containing integer computations with the unsigned shift-operator >>> give different results on the Java resp. the Java Card platform.

Closely related to our approach is Chalin's work [7]. He argues that the semantics of JML's arithmetic types (which are finite as in Java) diverges from the user's intuition. In fact, a high number of published JML specifications are shown to be inadequate due to that problem. As a solution, Chalin proposes an extension of JML with an infinite arithmetic type.

Structure of this Paper. The structure of this papers is as follows: After explaining in Section 2 why using only one of the two integer semantics (infinite as in UML/OCL resp. finite as in Java) is problematic, we explain our approach that is based on combining both semantics in Section 3. In Section 4, we describe the sequent calculus for the combined semantics, which has been implemented in the KeY system. Finally, in Section 5 we give an example for using our approach in software development.

Due to space restrictions, the proofs of the theorems given in the following are only sketched, they can be found in [21].

2 Disadvantages of Not Combining Finite and Infinite Integer Types

In this section we explain the mutual deficiencies of the two integer semantics when used separately.

2.1 Disadvantages of Using an Infinite Integer Type

A concrete implementation can be regarded as a refinement of a given specification where, in particular, the data types used in the specification are refined by concrete data types available in the implementation language. Following [14], we say that a (concrete) data type correctly refines an (abstract) data type if in all circumstances and for all purposes the concrete type can be validly used in place of the abstract one.

Considering OCL and JAVA this means that the primitive JAVA type `int` (`byte`, `short`, `long` could be used as well) is used to implement the specification type INTEGER. Obviously, this is not a correct data type refinement in general. For example, the formula $\forall x.x + 1 > x$ is valid with x of type INTEGER but is not valid if the type INTEGER is replaced with `int` because it holds $\mathtt{MAX\_int} + 1 \doteq \mathtt{MIN\_int}$ and thus $\mathtt{MAX\_int} + 1 < \mathtt{MAX\_int}$.

In the following, a semantics for JavaDL that treats JAVA integers as if they were correct refinements of INTEGER is called $\mathcal{S}_{OCL}$. This semantics $\mathcal{S}_{OCL}$, where overflow is totally disregarded, allows to verify programs that do not satisfy the specification, which is not just a disadvantage but unacceptable.

2.2 Disadvantages of Using a Finite Integer Type

A semantics that uses finite integer types and exactly corresponds to the semantics defined in the JAVA language specification [11] (and thus to the semantics implemented by the JVM) is called $\mathcal{S}_{Java}$ in the following. Using semantics $\mathcal{S}_{Java}$, the validity of the JavaDL proof obligations implies that all specified properties hold during the execution of the program on the JVM. Thus, at first sight $\mathcal{S}_{Java}$ seems to be the right choice. But there are also some drawbacks which are discussed in the following.

If a program is correct using semantics $\mathcal{S}_{Java}$ it shows the expected verified functional behaviour (black-box behaviour). However, overflow may occur during execution leading to a discrepancy between the developer's intention and the actual internal (white-box) behaviour of the program. As long as neither specification nor implementation are modified this discrepancy has no effect. However, in an ongoing software development process programs are often modified. Then, a wrong understanding of the internal program behaviour easily leads to errors that are hard to find, precisely because the program behaviour is not understood.

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. The program shows the expected black-box behaviour but the white-box behaviour likely differs from the developer's intention.

As mentioned in the introduction, we call such programs that satisfy their specification but lead to (unexpected) overflow during execution "incidentally" correct, because we assume that the white-box behaviour of the program is not understood. In our opinion "incidentally" correct programs should be avoided because they are a permanent source of error in the ongoing software development process.

The above problem does not arise directly from the semantics $\mathcal{S}_{Java}$ itself but rather from the semantic gap between the specification language UML/OCL and the implementation language JAVA. Thus, the same problem also occurs with other specification and implementation languages.

Another disadvantage of $\mathcal{S}_{Java}$ is the fact that formulas, that are intuitively valid in mathematics like $\forall x.\exists y.y > x$ are not valid anymore if x, y are of a built-in JAVA type, like e.g. `int`.

Furthermore, using semantics $\mathcal{S}_{Java}$ requires to reason about modulo arithmetic. This is more complicated than reasoning about integers because many simplification rules known from integer arithmetic cannot be applied to modulo arithmetic (for example, in modulo arithmetic $x + 1 > x$ cannot be simplified to *true*).

Our experience shows that many proof goals involving integer arithmetics (that remain after the rules of our JavaDL calculus have been applied to handle the program part of a proof obligation) can be discharged automatically by decision procedures for arithmetical formulas. In the KeY prover we make use of freely available implementations of arithmetical decisions procedures, like the Cooperating Validity Checker [24] and the Simplify tool (which is part of ESC/Java [8]). Both do *not* work for modulo arithmetics.

3 Combining Finite and Infinite Integer Types

3.1 The Idea

Basically, there are two possible approaches to proving that a particular JAVA program (with finite integer types) correctly refines a particular UML/OCL specification (with infinite integers types).

Firstly, one can show that the observable behaviour of the program meets the specification (whether overflow occurs or not), without checking explicitly that there is any particular relation between the integer types in the program resp. the specification. This amounts to using semantics $\mathcal{S}_{Java}$, which allows "incidentally" correct programs.

Secondly, one can show that whenever one of the arithmetical operations[4] $\circ \in \{\texttt{+}_T, \texttt{-}_T, \texttt{*}_T, \texttt{/}_T, \texttt{\%}_T\}$ on a type[5] $T \in \{\texttt{int}, \texttt{long}\}$ is invoked during the execution of a program, the following pre-condition is met, ensuring that no overflow occurs:

[4] Here, we do not consider the bit-wise logical and shift operations on integers, i.e., `~` (complement), `&` (and), `|` (or), `^` (xor), `<<` (left shift), `>>` (right shift), `>>>` (unsigned right shift). They may cause an overflow effect, but a programmer using bit-wise logical or shift operators can be assumed to be aware of the data type's bit-representation and, thus, of its finiteness.

[5] In JAVA arithmetical operators exist only for the types `int` and `long`. Arguments of type `byte` or `short` are automatically cast to `int` (or to `long` if one operand is of type `long`) by the JVM. This is called *promotion*.

$$\mathtt{MIN\_}T \leq x \mathbin{\hat{\circ}} y \leq \mathtt{MAX\_}T,$$

where $\hat{\circ}$ is the UML/OCL operation on Integer corresponding to the Java operation $\circ$. By checking this pre-condition, we establish that the Java types are only used to the extent that they indeed are a refinement of the UML/OCL types. This check cannot be done once and for all but has to be repeated for each particular Java program.

We use this second approach that truly combines the two types of integer semantics and avoids "incidentally" correct programs. The generation of proof obligations corresponding to instances of the above pre-condition is built into our verification calculus (Section 4).

With our approach to handling Java's integers, we fulfil the following three demands:

1. If the proof obligation for the correctness of a program is discharged, then the program indeed satisfies the specification. That is, the semantics of JavaDL and our calculus correctly reflect the actual Java semantics.
2. Programs that are merely "incidentally" correct (due to unintended overflow) *cannot* be proved to be correct, i.e., the problem is detected during verification.
3. Formulas like $\forall x.\exists y.y > x$ that are valid over the (infinite) integers (and, thus, are valid according to the user's intuition) remain valid in our logic.

3.2 A More Formal View

This section gives a formal definition of our semantics $\mathcal{S}_{KeY}$ for the Java integers that combines the advantages of (finite) Java and (infinite) UML/OCL integer semantics.

We extend Java by the additional primitive data types

$$\mathtt{arithByte}, \mathtt{arithShort}, \mathtt{arithInt}, \mathtt{arithLong},$$

which are called *arithmetical types* in contrast to the built-in types `byte`, `short`, `int`, and `long`. The new arithmetical types have an *infinite* range. They are, however, not identical to the mathematical integers (as used in $\mathcal{S}_{OCL}$) because the semantics of their operators in case of an "overflow" is different (in fact, it remains unspecified).

Note, that this extension of Java syntax is harmless and does not require an adaptation of the Java compiler. The additional types are only used during verification. Once a program is proved correct, they can be replaced with the corresponding built-in types (Corollary 1 in Section 3.3).

Definition 1. *Let p be a program containing arithmetical types. Then the program ptransf(p) is the result of replacing in p all occurrences of arithmetical types with the corresponding built-in* Java *types.*

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

An obvious difference between our semantics $\mathcal{S}_{KeY}$ and $\mathcal{S}_{OCL}$ resp. $\mathcal{S}_{Java}$ is that the signatures of the underlying programming languages differ, since $\mathcal{S}_{KeY}$ is a semantics for JAVA with arithmetical types whereas $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$ are semantics for standard JAVA.

Because of their infinite range, not all values of an arithmetical type are representable in the corresponding built-in type. There are program states[6] in JavaDL with $\mathcal{S}_{KeY}$ that do not correspond to any state reachable by the JVM. In the following, we call such states "unreal".

Definition 2. *A variable or an attribute that has an arithmetical type T is in* valid range *(in a certain state) iff its value val satisfies the inequations*

$$\texttt{MIN\_}T' \leq val \qquad and \qquad val \leq \texttt{MAX\_}T' \ ,$$

where T' is the built-in JAVA *type corresponding to T.*

Definition 3. *A JavaDL state s is called a* real state *iff all program variables and attributes with an arithmetical type are in valid range. Otherwise, s is called an* unreal state*.*

As already mentioned, both $\mathcal{S}_{KeY}$ and $\mathcal{S}_{OCL}$ have the same infinite domain. The crucial difference is in the semantics of the operators: If the values of the arguments of an operator application in $\mathcal{S}_{KeY}$ are in valid range but the (mathematical) result is not (i.e., overflow would occur if the arithmetical types were replaced with the corresponding built-in types), then the result of the operation is unknown; it remains unspecified. Otherwise, i.e., if the result is in valid range, it is the same as in $\mathcal{S}_{OCL}$. Technically this is achieved by defining that the result is calculated in the overflow case by invoking a method `overflow(x,y,op)` (the third parameter `op` is the operator that caused overflow and `x`,`y` are the arguments), whose behaviour remains unspecified (it does not even have to terminate).

The method `overflow` is not invoked if at least one argument of the operation is already out of valid range. In that case, the semantics of the operation in our semantics $\mathcal{S}_{KeY}$ is the same as in $\mathcal{S}_{OCL}$. This definition cannot lead to incorrect program behaviour because the program state before executing the operation is unreal and cannot be reached in an actual execution of the program.

The main reason for leaving the result of integer operations unspecified in the overflow case is that no good semantics for the overflow case exists, i.e., there is no reasonable implementation for the method `overflow`. In particular, the following two implementations that seem useful at first have major drawbacks:

– The method `overflow` throws an exception, does not terminate, or shows some other sort of "exceptional" behaviour. Then the semantics differs from the actual Java semantics (where an overflow occurs without an exception

[6] A program state assigns values (of the appropriate type) to local program variables, static fields, and the fields of all existing objects and arrays.

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

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	—	—	no
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

being thrown). This leads to the same problem as with semantics $\mathcal{S}_{OCL}$, i.e., programs whose actual execution does not satisfy the specification could be verified to be correct.

– The method `overflow` calculates the result in the same way as it is done in Java, including overflow. This leads to the same problem as with semantics $\mathcal{S}_{Java}$, i.e., “incidentally” correct programs could be verified to be correct.

The instance of $\mathcal{S}_{KeY}$ that results from using the latter of the above two implementations for `overflow` (instead of leaving it unspecified) is very similar to $\mathcal{S}_{Java}$. In the following, it is therefore called $\mathcal{S}_{Java'}$. While the problem that programs may be only “incidentally” correct remains with $\mathcal{S}_{Java'}$, it has an advantage over $\mathcal{S}_{Java}$: Using arithmetical types, formulas like $\forall x.\exists y.y > x$ are valid (other differences are discussed later).

Another reason for leaving `overflow` unspecified is that, if a JavaDL formula ϕ is derivable in our calculus for JavaDL based on $\mathcal{S}_{KeY}$ (i.e. `overflow` remains unspecified), then ϕ is valid for *all* implementations of `overflow` (this follows from the soundness of the calculus). In particular, one can conclude that (1) ϕ is valid in semantics $\mathcal{S}_{Java'}$ and (2) the validity of ϕ is not “incidental” (due to an overflow).

Example 1. The formula

$$\langle \texttt{j=i+1;} \rangle \texttt{j} \doteq \texttt{i} + 1$$

(where `i,j` are of an arithmetical type T) is *not* valid and not provable in our calculus (because `j=i+1` may cause an overflow after which $\texttt{j} \doteq \texttt{i} + 1$ does not hold).

However, the formula

$$\texttt{i} > \texttt{MAX\_}T \rightarrow \langle \texttt{j=i+1;} \rangle \texttt{j} \doteq \texttt{i} + 1$$

is valid in $\mathcal{S}_{KeY}$ and provable in our calculus. As explained above this is reasonable as the premiss $\texttt{i} > \texttt{MAX\_}T$ is never true during the actual execution of a Java program.

In $\mathcal{S}_{KeY}$, the semantics of the built-in types `byte`, `short`, `int`, `long` and the operators acting on them exactly corresponds to semantics $\mathcal{S}_{Java}$ and thus to the

Table 2. Validity of sample formulas in the different semantics.

	$\mathcal{S}_{OCL}$, $\mathbb{Z}$	$\mathcal{S}_{Java}$	$\mathcal{S}_{KeY}$	
			`arithInt`	`int`
Effects of overflow				
$\langle\texttt{i=MAX\_int+1;}\rangle\texttt{i} \doteq \texttt{MIN\_int}$		$\surd$		$\surd$
$\langle\texttt{i=0; while (i>=0) i++;}\rangle \mathit{true}$		$\surd$		$\surd$
$\exists\texttt{i}.(\texttt{i} > 0 \land \langle\texttt{i=i+1;}\rangle\texttt{i} < 0$		$\surd$		$\surd$
$\forall\texttt{i}.\langle\texttt{i=i+1;}\rangle\texttt{i} \doteq \texttt{i} + 1$	$\surd$			
Incidentally correct programs				
$\texttt{i} > 0 \to \langle\texttt{i=i+1; i=i-1;}\rangle\texttt{i} > 0$	$\surd$	$\surd$		$\surd$
$\forall\texttt{i}.\mathit{even}(\texttt{i}) \to \langle\texttt{i=i+2;}\rangle\mathit{even}(\texttt{i})$	$\surd$	$\surd$		$\surd$
Effects of unreal states				
$\forall\texttt{i}.\langle\texttt{i=i;}\rangle\texttt{i} \doteq \texttt{i}$	$\surd$	$\surd$	$\surd$	$\surd$
$\forall\texttt{i}.(\langle\texttt{j=i+1;}\rangle\texttt{j} \doteq \texttt{i} + 1)$	$\surd$			
$\forall\texttt{i}.(\texttt{i} > \texttt{MAX\_}T \to \langle\texttt{j=i+1;}\rangle\texttt{j} \doteq \texttt{i} + 1)$	$\surd$	$\surd$	$\surd$	$\surd$
$\forall\texttt{i}.(\texttt{i} > \texttt{MAX\_}T \to \mathit{false})$		$\surd$		$\surd$
Pure first-order formulas				
$\forall\texttt{i}.\texttt{i} + 1 > \texttt{i}$	$\surd$		$\surd$	
$\forall\texttt{i}.\exists\texttt{j}.\texttt{j} > \texttt{i}$	$\surd$		$\surd$	

definitions in the JAVA language specification. Hence, using the built-in types, it is still possible to make use of the effects of overflow by explicitly using the primitive built-in JAVA types in both the specification and the implementation.

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

Table 2 shows in which of the different semantics some sample formulas are valid. For $\mathcal{S}_{KeY}$, the cases that the program variables `i`, `j` are of type `arithInt` resp. `int` are distinguished.

3.3 Properties of the Combined Semantics

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, a program p, whose initial state is a real state, 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 important properties of $\mathcal{S}_{KeY}$. It states that, if the formula $\Gamma \to \langle p\rangle\psi$ is valid in $\mathcal{S}_{KeY}$ and the program p is started in a real state s such that $s \models_{\mathcal{S}_{KeY}} \Gamma$, no overflow occurs during the execution of the transformed program $\mathit{ptransf}(p)$ on the JAVA virtual machine, and after the execution the property ψ holds.

Note, that Corollary 1 does not apply to arbitrary formulas. For example, a formula of the form $[p]\mathit{true}$ is always derivable, whether overflow occurs during the execution of p or not. However, the generation of proof obligations for the correctness of a method typically results in formulas of the form $\Gamma \to \langle p\rangle\psi$ (Γ results from the pre-condition, ψ from the post-condition, and p is the implementation), so this is not a real restriction in practice.

The following theorems show that the differences between the verified program p and the actually executed program $\mathit{ptransf}(p)$ do not affect the verified behaviour of p.

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

Definition 4. *Let s be a real JavaDL state. The* isomorphic state $\mathit{iso}(s)$ *to s is the JVM state in which all state elements (program variables and fields) with an arithmetical type in s are of the corresponding built-in type and are assigned the same values as in s.*

If s is a real state, the existence of $\mathit{iso}(s)$ 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 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 $\mathit{iso}(s)\,[\![\mathit{ptransf}(p)]\!]_{\mathcal{S}_{Java}}\,\mathit{iso}(s')$.*

Corollary 1. *Let Γ, ψ be pure first-order predicate logic formulas, let p be an arbitrary* Java *program that may contain arithmetical types, and let s be an arbitrary JavaDL state.*

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

3.4 Variants of the Combined Semantics

In the definition of semantics $\mathcal{S}_{KeY}$, the method `overflow` remains unspecified. By giving a partial specification, i.e., axioms that `overflow` must satisfy, it is possible to define variants of $\mathcal{S}_{KeY}$. That way, one can allow certain occurrences of overflow, namely those which can be shown to be "harmless" using the additional axioms.

For example, one can define that the method `overflow` always terminates or implements an operation that is symmetric w.r.t. its arguments. If an axiom is added that `overflow` always terminates, a formula like $\Gamma \to \langle p \rangle \mathit{true}$ can be valid, even if overflow occurs during the execution of p, since goals of the form $\Gamma \vdash \langle$`x=overflow(arg1,arg2,op);`$\rangle \mathit{true}$ can immediately be closed using the information that the invocation of `overflow` terminates. That is, using such an axiom all overflow occurrences are defined to be "harmless" in cases where we are only interested in termination.

As long as the additional axioms are satisfiable by the instances $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java'}$ of $\mathcal{S}_{KeY}$, Theorem 2 and Theorem 3 still hold.

3.5 Steps in Software Development

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

1. *Specification:* In the UML/OCL specification, the 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 sequent calculus (see Section 4), one has to derive the proof obligations generated from the specification and implementation using the translation described in [5]. If all proof obligations are derivable, then Corollary 1 implies that the program, if the requirements of the corollary are satisfied and after replacing the arithmetical types with the corresponding built-in types, satisfies all specified properties during the execution on the JVM and in particular, no overflow occurs.

4 Sequent Calculus for the Combined Semantics

4.1 Overview

As already explained, the KeY system's deduction component uses the program logic JavaDL, which is a version of Dynamic Logic modified to handle Java Card programs [4]. We have extended and adapted that calculus to implement our approach to handling integer arithmetic using the semantics $\mathcal{S}_{KeY}$.

Here, we cannot list all rules of the adapted calculus (they can be found in [21]). To illustrate how the calculus works, we present some typical rules representing the two different rule types: program transformation rules to evaluate compound Java expressions (Section 4.4) and rules to symbolically execute simple Java expressions (Section 4.5).

The semantics of the rules is that, if the premisses (the sequent(s) at the top) are valid, then the conclusion (the sequent at the bottom) is valid. In practice, rules are applied from bottom to top: from the old proof obligation, new proof obligations are derived.

Sequents are notated following the scheme

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

which has the same semantics as the formula

$$(\forall x_1) \cdots (\forall x_k)((\phi_1 \wedge \ldots \wedge \phi_m) \to (\psi_1 \vee \ldots \vee \psi_n)) \ ,$$

where $x_1, \ldots, x_k$ are the free variables of the sequent.

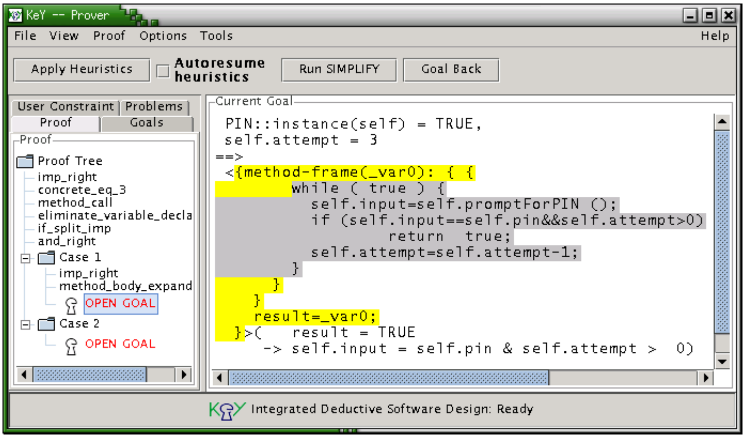
4.2 Notation for Rule Schemata

In the following rule schemata, *var* is a local program variable (of an arithmetical type) whose access cannot cause side-effects. For expressions that potentially have side-effects (like, e.g., an attribute access that might cause a `NullPointerException`) the rules cannot be applied and other rules that evaluate the complex expression and assign the result to a new local variable have to be applied first. Similarly, *simp* satisfies the restrictions on *var* as well or it is an integer literal (whose evaluation is also without side-effects). There is no restriction on *expr*, which is an arbitrary Java expression of a primitive integer type (its evaluation may have side-effects).

The predicate $in_T(x)$ expresses that x is in valid range, i.e.,

$$in_T(x) \quad\equiv\quad \texttt{MIN\_}T \leq x \wedge x \leq \texttt{MAX\_}T \ .$$

The rules of our calculus operate on the first *active* statement p of a program $\pi\, p\, \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`method-frame(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active statement is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately. The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the Java block "`l:{try{ i=0; j=0; }finally{ k=0; }}`", operating on its first active statement "`i=0;`", then the non-active prefix π is "`l:{try{`" and the "rest" ω is "`j=0; }finally{ k=0; }}`". Prefix, active statement, and postfix are automatically highlighted in the KeY prover as shown in Figure 1.



```
 PIN::instance(self) = TRUE,
 self.attempt = 3
==>
 <{method-frame(_var0): { {
         while ( true ) {
           self.input=self.promptForPIN ();
           if (self.input==self.pin&&self.attempt>0)
                   return  true;
           self.attempt=self.attempt-1;
         }
       }
     }
     result=_var0;
  }>(    result = TRUE
    -> self.input = self.pin & self.attempt >  0)
```

Fig. 1. KeY prover window with the proof obligation generated from the example.

4.3 State Updates

We allow *updates* of the form $\{x := t\}$ resp. $\{o.a := t\}$ to be attached to terms and formulas, where x is a program variable, o is a term denoting an object with attribute a, and t is a term (which cannot have side-effects). The intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\{x := t\}\phi$ has the same semantics as $\langle \texttt{x = t;} \rangle\phi$.

4.4 Program Transformation Rules

The Rule for Postfix Increment. This rule transforms a postfix increment into a normal JAVA addition.

$$\frac{\Gamma \ \vdash \ \langle \pi \ \mathit{var} \ \texttt{=} \ \texttt{(}T\texttt{)(}\mathit{var} \ \texttt{+ 1);} \ \omega\rangle\phi, \ \Delta}{\Gamma \ \vdash \ \langle \pi \ \mathit{var}\texttt{++;} \ \omega\rangle\phi, \ \Delta} \tag{R1}$$

T is the (declared) type of *var*. The explicit type cast is necessary since the arguments of `+` are *promoted* to `int` or `long` but the postfix increment operator `++` does not involve promotion.

The Rule for Compound Assignment. This rule transforms a statement containing the compound assignment operator `+=` into a semantically equivalent statement with the simple assignment operator `=` (again, T is the declared type of *var*).

$$\frac{\Gamma \ \vdash \ \langle \pi \ \mathit{var} \ \texttt{=} \ \texttt{(}T\texttt{)(}\mathit{var} \ \texttt{+} \ \mathit{expr}\texttt{);} \ \omega\rangle\phi, \ \Delta}{\Gamma \ \vdash \ \langle \pi \ \mathit{var} \ \texttt{+=} \ \mathit{expr}\texttt{;} \ \omega\rangle\phi, \ \Delta} \tag{R2}$$

For the soundness of both (R1) and (R2), it is essential that *var* does not have side-effects because *var* is evaluated twice in the premisses and only once in the conclusions.

4.5 Symbolic Execution Rules

For the soundness of the following three rules it is important that *var* and *simp* are of an arithmetical type (rules for the built-in types can be found in [21]).

The Rule for Type Cast to an Arithmetical Type. A type cast from the declared type S of *simp* to an arithmetical type T causes overflow if the value of *simp* is in valid range of S but not in valid range of T (second premiss).

$$\frac{\begin{array}{l}\Gamma, \ \mathit{in}_S(\mathit{simp}) \to \mathit{in}_T(\mathit{simp}) \ \vdash \ \{\mathit{var} := \mathit{simp}\}\langle \pi \ \omega\rangle\phi, \ \Delta \\ \Gamma, \ \mathit{in}_S(\mathit{simp}), \ \neg \mathit{in}_T(\mathit{simp}) \ \vdash \\ \qquad \langle \pi \ \mathit{var} \ \texttt{=} \ \texttt{overflow(}\mathit{simp}\texttt{, "cast(}T\texttt{)");} \ \omega\rangle\phi, \ \Delta\end{array}}{\Gamma \ \vdash \ \langle \pi \ \mathit{var} \ \texttt{=} \ \texttt{(}T\texttt{)}\mathit{simp}\texttt{;} \ \omega\rangle\phi, \ \Delta} \tag{R3}$$

The Rule for Unary Minus. The unary minus operator only causes overflow if the value of *var* is equal to $\mathtt{MIN\_}T$ (where T is the promoted type of *var*).

$$\frac{\begin{array}{l} \Gamma,\ \neg simp \doteq \mathtt{MIN\_}T \ \vdash\ \{var := -simp\}\langle \pi\ \omega\rangle\phi,\ \Delta \\ \Gamma,\ simp \doteq \mathtt{MIN\_}T \ \vdash\ \langle \pi\ var\ \mathtt{=}\ \mathtt{overflow(}simp\mathtt{,"-");}\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma\ \vdash\ \langle \pi\ var\ \mathtt{=}\ \mathtt{-}simp\mathtt{;}\ \omega\rangle\phi,\ \Delta} \quad \text{(R4)}$$

The Rule for Subtraction. This rule symbolically executes a subtraction and checks for possible overflow.

$$\frac{\begin{array}{l} \Gamma,\ in_{T_1}(simp_1) \wedge in_{T_2}(simp_2) \rightarrow in_T(simp_1 - simp_2)\ \vdash \\ \qquad\qquad \{var := simp_1 - simp_2\}\langle \pi\ \omega\rangle\phi,\ \Delta \\ \Gamma,\ in_{T_1}(simp_1),\ in_{T_2}(simp_2),\ \neg in_T(simp_1 - simp_2)\ \vdash \\ \qquad\qquad \langle \pi\ var\ \mathtt{=}\ \mathtt{overflow(}simp_1\mathtt{,}simp_2\mathtt{,"-");}\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma\ \vdash\ \langle \pi\ var\ \mathtt{=}\ simp_1\ \mathtt{-}\ simp_2\mathtt{;}\ \omega\rangle\phi,\ \Delta} \quad \text{(R5)}$$

The first premiss applies in case (1) both arguments and the result are in valid range (no overflow can occur) and (2) one of the two arguments is not in valid range (overflow is “allowed” as the initial state is already an unreal state).

The second premiss states that, if the arguments are in valid range but the result is not, the result of the arithmetical Java operation is calculated by the unspecified method `overflow`.

5 Extended Example

In this example we describe the specification, implementation, and verification of a PIN-check module for an automated teller machine (ATM). Before we give an informal specification, we describe the scenario of a customer trying to withdraw money.

After inserting the credit card, the user is prompted for his PIN. If the correct PIN is entered, the customer may withdraw money and then gets the credit card back. Otherwise, if the PIN is incorrect, two more attempts are left to enter the correct PIN. When an incorrect PIN has been entered more than two times, it is still possible to enter more PINs but even if one of these PINs is correct, no money can be withdrawn and the credit card is retained to prevent misuse.

Our PIN-check module contains a boolean method `pinCheck` that checks whether the PIN entered is correct and the number of attempts left is greater than zero. The informal specification of this method is, that the result value is `true` only if the PIN entered is correct and the number of attempts left is positive (it is decreased after unsuccessful attempts).

The formal specification of the method `pinCheck` consists of the OCL pre-/ post-conditions

```
context PIN::pinCheck(input:Integer):Boolean
pre:  attempt=3
post: result=true implies input=pin and attempt>0
```

stating, under the assumption that `attempt` is equal to three in the pre-state, that `input` (the PIN entered) is equal to `pin` (the correct PIN of the customer) and the number of attempts left is greater than zero if the return value of `pinCheck` is `true`.

In this example the above formal specification is not complete (with respect to the informal specification):[7] The relation between the attribute `attempt` and the actual number of attempts made to enter the PIN (invocations of the method `promptForPIN`) is not specified. The implicit assumption is that the number of attempts made equals 3 − `attempt`. As we will see however, this assumption does not hold any more when decreasing `attempt` causes (unintended) overflow—leading to undesired results.

Without our additional arithmetical types, a possible implementation of the method `pinCheck` is the one shown on the left in Figure 2. Such an implementation may be 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 the method `promptForPin` is invoked. It returns the PIN entered by the user, which is then assigned to the variable `input`. In case the entered PIN is equal to the user's correct PIN and the number of attempts left is greater than zero, the loop and thus the method terminates with "`return true;`". Otherwise, the variable `attempt`, counting the attempts left, is decreased by one.

The generation of proof obligations from the formal OCL specification and the implementation yields the following JavaDL formula, where the body of method `pinCheck` is abbreviated with p:

$$\texttt{attempt} \doteq 3 \ \vdash \ \langle p \rangle (\texttt{result} \doteq \mathit{true} \rightarrow \texttt{input} \doteq \texttt{pin} \land \texttt{attempt} > 0)$$

Figure 1 shows this sequent after "unpacking" the method body of `pinCheck` in the KeY prover. 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 implementation can be said to be correct in the sense that it satisfies the specification.

But this implementation has an unintended behaviour. Suppose the credit card has been stolen and the thief wants to withdraw money but does not know the correct PIN. Thus, he has to try all possible PINs. According to the informal specification, after three wrong attempts any further attempt should not be successful any more. But if the thief does not give up, at some point the counter `attempt` will overflow and get the positive value `MAX_int`. Then, the thief has many attempts to enter the correct PIN and thus, to withdraw money.

[7] In this simple example, the incompleteness of the specification may easily be uncovered but in more complex cases it is not trivial to check that the formal specification really corresponds to the informal specification.

```
class PIN {
 private int pin=1234;
 private int attempt;
 int input;

 public boolean pinCheck() {
  while (true) {
   input=promptForPIN();
   if (input==pin && attempt>0)
    return true;

   attempt=attempt-1;

  }
 }
}
```

```
class PIN {
 private int pin=1234;
 private arithInt attempt;
 int input;

 public boolean pinCheck() {
  while (true){
   input=promptForPIN();
   if (input==pin && attempt>0)
    return true;

   if (attempt>0)
     attempt=attempt-1;
  }
 }
}
```

Fig. 2. Implementation of method `pinCheck` without (left) and with (right) using the additional arithmetical type `arithInt`.

The main reasons for this unexpected behaviour are the incomplete formal specification and the implementation that is "incidentally" correct w.r.t. the formal specification. In the following, we demonstrate that this unintended behaviour of the program can be detected following our approach.

In the implementation, we now 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`.

We do not show single proof steps and the corresponding rules that have to be applied. However, the crucial point in the proof is when it comes to handle the statement "`attempt=attempt-1;`". After applying rule (R5) one of the new goals is the following:

$$\mathit{in}_{\texttt{int}}(\texttt{attempt}),\ \mathit{in}_{\texttt{int}}(1),\ \neg\mathit{in}_{\texttt{int}}(\texttt{attempt}-1)\ \vdash \\ \langle\texttt{attempt = overflow(attempt,1,"-");}\rangle\phi.$$

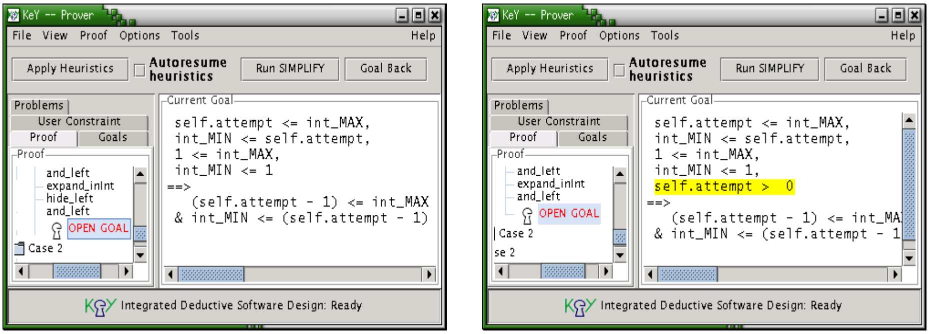
Since nothing is known about `overflow`, the only way to derive this in our JavaDL calculus is to prove—as a lemma or sub-goal—that no overflow occurs (and, thus, `overflow` is not invoked). Thus, one has to derive

$$\mathit{in}_{\texttt{int}}(\texttt{attempt}),\ \mathit{in}_{\texttt{int}}(1),\ \neg\mathit{in}_{\texttt{int}}(\texttt{attempt}-1)\ \vdash\ \mathit{false}$$

or equivalently

$$\mathit{in}_{\texttt{int}}(\texttt{attempt}),\ \mathit{in}_{\texttt{int}}(1)\ \vdash\ \mathit{in}_{\texttt{int}}(\texttt{attempt}-1)\ .$$

But the above sequent is neither valid nor derivable, because it is not true in states where `attempt` has the value `MIN_int`. In such states the subtraction causes overflow and the sequent does not hold because its left side is true but its right side is false (as `attempt` $-$ 1 is not in valid range). The left part of Figure 3 shows the invalid sequent in the KeY prover.



```
self.attempt <= int_MAX,
int_MIN <= self.attempt,
1 <= int_MAX,
int_MIN <= 1
==>
   (self.attempt - 1) <= int_MAX
 & int_MIN <= (self.attempt - 1)
```

```
self.attempt <= int_MAX,
int_MIN <= self.attempt,
1 <= int_MAX,
int_MIN <= 1,
self.attempt >  0
==>
   (self.attempt - 1) <= int_MA
 & int_MIN <= (self.attempt - 1
```

Fig. 3. The KeY prover window on the left shows an invalid sequent. The prover window on the right shows the same sequent with the additional highlighted premiss that makes the sequent valid.

Note, that this error is uncovered by using our additional arithmetical types and our semantics $\mathcal{S}_{KeY}$. If the built-in type `int` is used in the implementation, the error is not detected.

Since the proof obligation is not derivable in our calculus, one has to correct the implementation to be able to prove its correctness. For example, one can add a check whether the value of `attempt` is greater than 0 before it is decremented. This results in the implementation depicted on the right side in Figure 2. Trying to verify this new implementation with the KeY system leads to the sequent shown in the right part of Figure 3. In contrast to the one shown in the left part of Figure 3, this sequent is valid because of the additional formula (`self.attempt`) > 0 on the left side, which stems from the added check in the revised implementation.

The resulting proof obligation can now be derived in our calculus and, thus, Corollary 1 implies that no overflow occurs if the type `arithInt` is replaced with `int` in order to execute the program on the Java virtual machine. With the improved 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 (incomplete) specification, which is satisfied by the first implementation. Due to unintended overflow, this implementation has a behaviour not intended by the programmer. Following our approach, the 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. Thus, if a program containing arithmetical types cannot be verified due to overflow, it should always be checked whether the specification is adequate (it may be based on implicit assumptions that should be made explicit).

6 Conclusion

We have presented a method for handling the data refinement relation between infinite and finite integer types. The main design goals of our approach are:

- "incidentally" correct programs are avoided by ensuring that no overflow occurs,
- the handling of the refinement relation is integrated into the verification calculus and, thus, hidden from the user as far as possible,
- the semantics combining both finite and infinite JAVA types provides a well-defined theoretical basis for our approach.

Acknowledgement. We thank R. Bubel, A. Roth, P. H. Schmitt, and the anonymous referees for important feedback on drafts of the paper.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, pages 1–42, 2004. To appear.
3. 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.
4. 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.
5. 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. Available at: `http://www.key-project.org/doc/2002/BeckertKellerSchmitt02.ps.gz`.
6. N. Cataño and M. Huisman. Formal specification and static checking of Gemplus' electronic purse using ESC/Java. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings, FME 2002: Formal Methods - Getting IT Right, Copenhagen, Denmark*, LNCS 2391, pages 272–289. Springer, 2002.
7. P. Chalin. Improving JML: For a Safer and More Effective Language. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings, FME 2003: Formal Methods, Pisa, Italy*, LNCS 2805, pages 440–461. Springer, 2003.

8. ESC/Java (Extended Static Checking for Java). http://research.compaq.com/SRC/esc/.
9. European Space Agency. Ariane 501 inquiry board report, July 1996. Available at: http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf.
10. S. Glesner. Java Card Integer Arithmetic: About an Inconsistency and Its Algebraic Reason, 2004. Draft.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.
12. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs), Nice, France*, LNCS 1690, pages 113–130. Springer, 1999.
13. J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS 1869, pages 234–251. Springer, 2000.
14. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, volume LNCS 213, pages 187–196. Springer, 1986.
15. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.
16. B. Jacobs. Java's Integral Types in PVS. In E. Najim, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pages 1–15. Springer, 2003.
17. A. B. Kit. Available at: http://www.gemplus.com/smart/r_d/publications/case-study/.
18. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
19. Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.
20. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential java. In S. D. Swierstra, editor, *Proceedings, European Symposium on Programming (ESOP), Amsterdam, The Netherlands*, LNCS 1576, 1999.
21. S. Schlager. Handling of Integer Arithmetic in the Verification of Java Programs. Master's thesis, Universität Karlsruhe, 2002. Available at: http://www.key-project.org/doc/2002/DA-Schlager.ps.gz.
22. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
23. K. Stenzel. Verification of JavaCard Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available at: http://www.informatik.uni-augsburg.de/swt/fmg/papers/.
24. A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.
25. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.