To appear in Formal Aspects of Computing

Refinement and Retrenchment for Programming Language Data Types

Bernhard Beckert[1] and Steffen Schlager[2]

[1]University of Koblenz-Landau
Institute for Computer Science
D-56070 Koblenz, Germany
beckert@uni-koblenz.de
[2]Universität Karlsruhe
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
schlager@ira.uka.de

Abstract. Refinement is a well-established and accepted technique for the systematic development of correct software systems. However, for the step from already refined specification to implementation, a correct refinement is often not possible because the data types used in the specification respectively the implementation language differ. In this paper, we discuss this problem and its consequences, using the integer data types of Java as an example, which do not correctly refine the mathematical integers $\mathbb{Z}$. We present a solution, which can be seen as a generalisation of refinement and a variant of retrenchment. It has successfully been implemented as part of the KeY software verification system.

Keywords: software verification, formal specification, retrenchment, refinement, Java, UML/OCL, integer arithmetic.

1. Introduction

The idea of refinement. Refinement is a well-established and accepted technique for the systematic development of correct software systems. Starting from an initial formal specification, refinement steps are performed to obtain a correct implementation—a software system satisfying the specification. Each refinement step may add more details, e.g. by removing non-determinism.

To guarantee correctness of the system being developed, the refinement steps themselves must be correct, i.e., they must adhere to certain rules. Then, the refinement process preserves all properties of the abstract system. They are *automatically* satisfied by the concrete system as well; it is not necessary to re-prove them. Only those aspects and details of the concrete system have to be verified that are not present in the abstract one.

Correspondence and offprint requests to: B. Beckert and S. Schlager

The goal of refinement is to eventually obtain a software system that is correct by construction as only (correct) refinement steps are performed from specification to code.

In this paper, we concentrate on the refinement of data types (data refinement), in particular the refinement of integer data types. Following [HHS86], a data refinement is correct if in all circumstances and for all purposes the concrete data type can be validly used in place of the abstract one.

The problem. In practice, constructing correct refinement steps is not always easy—and sometimes not even possible. A crucial situation where a correct *data* refinement is often not possible is the step from an already refined specification to implementation code. The problem is that this (last) step involves a switch from the specification language to the implementation language (the earlier refinement steps stay within the specification language).

In particular, specification languages usually offer integer data types with an infinite domain, for which the finite data types in programming languages are not a correct refinement.

Consider, for example, the following JAVA method, which—supposedly—implements an operation computing the sum of two integers:

```
int sum(int x, int y) {
  return x+y;
}
```

At first sight, this method seems to satisfy its specification, i.e., to return the *mathematical* sum of the parameter values. In truth, however, the implementation is *not* correct. The reason is that the (finite) JAVA type `int` used in the implementation does not correctly refine the (infinite) specification type $\mathbb{Z}$. In particular, the JAVA operation $+_{\texttt{int}}$ is not a refinement of the operation $+_{\mathbb{Z}}$, because it computes the sum of the two arguments *modulo the size of the type* `int`, e.g.:

$$\begin{array}{llcrl} 2147483647 & +_{\mathbb{Z}} & 1 = & 2147483648 & \text{but} \\ 2147483647 & +_{\texttt{int}} & 1 = & -2147483648 & \end{array}$$

Possible solutions. There are basically three possibilities to overcome the problem outlined above:

1. Changing the data type operations on the implementation level, such that the refinement becomes correct. This amounts to adapting the implementation data type to the specification type.
2. Changing the data type operations on the specification levels (including abstract ones), such that the refinement becomes correct. This amounts to adapting the specification data type to the implementation type.
3. Allowing a *limited* and *controlled* incorrectness in the "refinement" steps.

In the following, we explain why the third possibility is the best solution (although at first sight it may seem to jeopardise correctness), and we describe how it can be used in such a way that correctness of the overall software verification process is preserved.

The first two possibilities have serious drawbacks. On the one hand, changing and adapting the implementation data type (e.g., using long number arithmetics instead of the built-in integer types), introduces unnecessary and serious inefficiencies into the implementation. Moreover, such data type implementations are not always readily available. On the other hand, using an implementation language data type on the specification level—this approach is, for example, pursued in the JAVA Modeling Language (JML) [LBR99]—contradicts the idea that specifications should be abstract and hide implementation details. Humans think in terms of infinite (mathematical) types. It is, thus, not surprising that quite a number of JML specifications are inadequate, i.e. do not have the intended meaning [Cha03].[1] Inadequate specifications are expensive to fix since they are at the root of the development process. Moreover, some implementation details may not even be known during specification, e.g. the implementation language or the concrete data types that will be used. Obviously, an early specification containing such details is neither reusable nor comprehensive and hence more likely to be inadequate.

[1] To solve this problem, Chalin [Cha03] proposes to extend the JML, which does not support infinite integer types, with a type `infint` with infinite range. That, however, introduces into JML the problem of "incorrect refinement".

Our approach to solving the problem. The price we have to pay when we use the third of the above possibilities, i.e., allowing a limited incorrectness in the “refinement” steps, is that the implementation is not any more automatically correct “by refinement.” Rather, additional proofs are required. This approach, which can be seen as a generalisation of refinement, is an instance of Banach and Poppleton’s *retrenchment* paradigm [BP98].

The advantage of casting non-refinement steps into the retrenchment framework is that it becomes explicit where exactly the refinement conditions are violated and, thus, where correctness cannot be shown once and for all (as it is the case with correct refinement). Instead, we show the correctness of a program containing retrenchment by individually verifying critical situations. Note, that these additional proofs are still done at proof time. After these proofs have been done, no run-time checks are required.

In the following, we dicuss two instances of this approach:

- Cases for which the refinement is not correct are shown not to occur (this amounts to strengthening the preconditions for the invocation of the “refined” data type operations).
  Considering as an example the operation $+_{\mathtt{int}}$ respectively $+_{\mathbb{Z}}$, one has to show for each individual invocation of $x + y$ that the result does not exceed the (finite) range of JAVA `int`, i.e., does not lead to an overflow.
- It is shown that, for those cases where the refinement is not correct, the “refined” data type operations are sufficient (this amounts to weakening the postcondition).
  For the example of $+$, one has to show for the particular situation that using $x +_{\mathtt{int}} y$ instead of $x +_{\mathbb{Z}} y$ does not lead to incorrectness.

We will argue that the first of the above two possibilities is the better choice. It is implemented in the KeY system, which is an integrated tool for specification and verification of JAVA programs.

In the KeY system’s calculus, the exclusion of non-refinement cases for integer operations results in additional proof obligations stating that the result of an operation does not exceed the (finite) range of the expression type. 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 (the non-refinement cases do not occur). As said above, this check cannot be done once and for all, as it is the case for a correct refinement, but has to be repeated for each particular program. Since this is tedious and error-prone if done by hand we have integrated the generation of the additional proof obligation into our verification calculus. Most of these proof obligations are discharged automatically by our prover.

Remark on languages. 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 the approach we present in this paper. The languages UML/OCL and JAVA can be substituted by almost any other specification and implementation languages (e.g. Z [Spi92] or B [Abr96] respectively C++). We use UML/OCL and JAVA mainly because the work presented here has been carried out as part of the KeY project (see Sect. 4.1).

Also, note, that although we concentrate on the JAVA type `int`, everything said in the following applies just as well to the other JAVA integer types `byte`, `short`, and `long`.

Structure of this paper. The paper is organised as follows. In Sect. 2, we formally define and describe the notion of data refinement and the more general data retrenchment. In Sect. 3, we apply our retrenching approach to the particular case of integer arithmetic. Our implementation as part of the KeY system is presented in Sect. 4, including a description of the arithmetic rules used in the KeY program verification calculus. Sect. 5 contains an extended example; and finally, in Sect. 6, we discuss related work.

2. Refinement and Retrenchment

2.1. Basic Definitions

Operations are relations on states and input/output values. We do not define what a state is, because the internal structure of states is not relevant for the approach presented in this paper.

Definition 1. (Operation, Termination) Let S be a set of states, *Input* a set of possible inputs, and *Output* a set of possible outputs. An *operation*

$$Op \ \subseteq \ S \times \mathit{Input} \times S \times \mathit{Output}$$

is a relation on states and input/output values. We write $s[\![out = Op(in)]\!]s'$ iff $(s, in, s', out) \in Op$.

Operation Op started in a state $s \in S$ with an input *in terminates* iff there exists a state $s' \in S$ and an output *out* such that $s[\![out = Op(in)]\!]s'$. This is denoted by $s \models Op(in)\!\downarrow$. □

Without loss of generality, we only consider operations that have an output. Operations without output are considered to return an arbitrary value. For non-deterministic programming languages the definition of termination expresses that there is a possibility for the operation to terminates, whereas for a deterministic language the operation always terminates.

Definition 2. (Abstract Data Type) An *abstract data type (ADT)* is a 4-tuple $(S, V, \mathit{Init}, \{Op_i\}_{i \in I})$, where S is a set of states, $\mathit{Init} \subseteq S$ is an initialisation operation, and Op_i $(i \in I)$ are operations on S and the set $V = \mathit{Input} = \mathit{Output}$ of input/output values. □

For ADTs we only allow operations that have a single input and a single output value, and we assume that all operations of an ADT have the same input and output type. That is not a real restriction, since the set $V = \mathit{Input} = \mathit{Output}$ can be defined to be the union of all input and output types of the individual operations. The initialisation operation of an ADT is a unary predicate defining its sets of initial states.

Definition 3. (Operation Specification) Given sets S of states, *Input* of input, and *Output* of output values, an *operation specification* is a pair $(\mathit{Pre}, \mathit{Post})$ of predicates

$$\mathit{Pre} \subseteq S \times \mathit{Input} \ \text{ and } \ \mathit{Post} \subseteq S \times \mathit{Input} \times S \times \mathit{Output} \ .$$

□

Definition 4. (Operation Correctness) Given sets S of states, *Input* of input, *Output* of output values, and an operation specification $(\mathit{Pre}, \mathit{Post})$. An operation $Op \subseteq S \times \mathit{Input} \times S \times \mathit{Output}$ is *correct* w.r.t. $(\mathit{Pre}, \mathit{Post})$ iff, for all $(s, in, s', out) \in Op$,

$$(s, in) \in \mathit{Pre} \ \text{ implies } \ (s, in, s', out) \in \mathit{Post} \ .$$

□

For a more concise presentation, we use the following notation conventions: A variable s_{abstr} always ranges over a set S_{abstr} of states. The same holds analogously for s_{concr} and s. The input value *in* and the output value *out* range over *Input* and *Output*, respectively. A subscripted variable like in_{abstr} ranges over Input_{abstr}, etc.

2.2. Refinement

As explained in the introduction, the basic idea of refinement is quite simple and independent of a particular computational formalism. It is based on the principle of substitutivity which states that it is acceptable to replace an operation by another operation as long as it is impossible for the environment to observe the substitution. In this paper we follow the definition of refinement given by Boiten and Derrick [DB01].

2.2.1. Data Refinement

Data refinement is a relation between a (concrete) operation and another (abstract) operation, where the concrete operation is supposed to simulate the abstract one. These operations are associated with ADTs that may differ in their sets S_{concr} respectively S_{abstr} of state spaces. They must, however, have the same sets of input/output values.

Refinement requires three conditions to be met:

- The *initialisation condition* requires that every initial state of the concrete type must be related to some initial state of the abstract type.

- The concrete operation must not terminate in any state that is inconsistent with the behaviour of the abstract operation (*correctness condition*). This implies an elimination of non-determinism or a stronger postcondition.

Definition 5. (Forward Simulation Data Refinement) Let

$$\mathcal{A} = (S_{abstr}, V, \mathit{Init}_{abstr}, \{\mathit{Op}_{abstr,i}\}_{i \in I}) \text{ and } \mathcal{C} = (S_{concr}, V, \mathit{Init}_{concr}, \{\mathit{Op}_{concr,i}\}_{i \in I})$$

be ADTs with the same sets $\mathit{Input} = \mathit{Output} = V$ for all operations $\mathit{Op}_{abstr,i}$ and $\mathit{Op}_{concr,i}$.

The ADT $\mathcal{C}$ is a *data refinement* of the ADT $\mathcal{A}$ via a *retrieve relation* $R \subseteq S_{abstr} \times S_{concr}$ iff the following holds:

Initialisation: For every state s_{concr},

if $\mathit{Init}_{concr}(s_{concr})$,
then there exists a state s_{abstr} with $\mathit{Init}_{abstr}(s_{abstr})$ and $R(s_{abstr}, s_{concr})$.

Correctness: For all states $s_{abstr}, s_{concr}, s'_{concr}$, all input values *in*, and all output values *out*,

if $s_{abstr} \models \mathit{Op}_{abstr}(\mathit{in})\!\downarrow$ and $R(s_{abstr}, s_{concr})$ and $s_{concr}[\![\mathit{out} = \mathit{Op}_{concr}(\mathit{in})]\!]s'_{concr}$,
then there exists a state s'_{abstr} with $s_{abstr}[\![\mathit{out} = \mathit{Op}_{abstr}(\mathit{in})]\!]s'_{abstr}$ and $R(s'_{abstr}, s'_{concr})$.

□

The above version of refinement only works for partial correctness. If termination properties are to be preserved as well, an additional *applicability condition* has to be used ensuring that the concrete operation is defined on all states on which the abstract operation is defined:

if $s_{abstr} \models \mathit{Op}_{abstr}(\mathit{in})\!\downarrow$ and $R(s_{abstr}, s_{concr})$,
then $s_{concr} \models \mathit{Op}_{concr}(\mathit{in})\!\downarrow$.

There are several other versions and variants of refinement. In IO refinement[BD98], for example, the sets of input and output values of the abstract respectively the concrete operation are allowed to differ (one may want to use, e.g., binary numbers for abstract inputs and decimal numbers for concrete inputs).

2.2.2. Operation Refinement

In the following, we concentrate on a restricted version of refinement, called *operation refinement*, where the concrete and the abstract operation are associated with the same ADT and, thus, the same state space.

Definition 6. (Operation Refinement) An operation Op_{concr} is an *operation refinement* of an operation Op_{abstr} (over the same state space S) iff, for all states s, s', all input values *in*, and all output values *out*,

if $s \models \mathit{Op}_{abstr}(\mathit{in})\!\downarrow$ and $s[\![\mathit{out} = \mathit{Op}_{concr}(\mathit{in})]\!]s'$ then $s[\![\mathit{out} = \mathit{Op}_{abstr}(\mathit{in})]\!]s'$.

□

Theorem 1. (Correctness w.r.t. Refinement) Let $\mathit{Op}_{abstr}, \mathit{Op}_{concr}$ be operations over the same state space, and let $(\mathit{Pre}, \mathit{Post})$ be a specification.

If

- Op_{abstr} is correct with respect to $(\mathit{Pre}, \mathit{Post})$ and
- Op_{concr} is an operation refinement of Op_{abstr},

then Op_{concr} is correct with respect to $(\mathit{Pre}, \mathit{Post})$. □

Proof. Trivial. □

2.3. Retrenchment

Retrenchment was devised by R. Banach and M. Poppleton [BP98] as a generalisation of refinement. Their motivation was the observation that refinement and even liberalisations thereof, like IO refinement [BD98], are too restrictive for most realistic system developments. For example, real numbers cannot be refined to floating point numbers, and the (mathematical) integers cannot be refined to JAVA integers (see Sect. 1).

The formal definition of retrenchment uses, in addition to a retrieve relation R (as in data refinement), two more relations, called *within* W and *concedes* C. The within relation W is used to limit the set of states and inputs for which the relationship between abstract and concrete operations needs to be established. The concedes clause deals with the case where W holds but the retrieve relation R between abstract and concrete post state cannot be established.

Definition 7. (Retrenchment) Let

$$\mathcal{A} = (S_{abstr}, V_{abstr}, \mathit{Init}_{abstr}, \{Op_{abstr,i}\}_{i\in I}) \text{ and } \mathcal{C} = (S_{concr}, V_{concr}, \mathit{Init}_{concr}, \{Op_{concr,i}\}_{i\in I})$$

be ADTs with $\mathit{Input}_{abstr} = \mathit{Output}_{abstr} = V_{abstr}$ for all operations $Op_{abstr,i}$ and $\mathit{Input}_{concr} = \mathit{Output}_{concr} = V_{concr}$ for all operations $Op_{concr,i}$.

The ADT $\mathcal{C}$ is a *retrenchment* of the ADT $\mathcal{A}$ via

- a *retrieve* relation $R \subseteq S_{abstr} \times S_{concr}$,
- a *within* relation $W \subseteq \mathit{Input}_{abstr} \times \mathit{Input}_{concr} \times S_{abstr} \times S_{concr}$, and
- a *concedes* relation $C \subseteq S_{abstr} \times S_{concr} \times S_{abstr} \times S_{concr} \times \mathit{Output}_{abstr} \times \mathit{Output}_{concr}$

iff the following holds:

Initialisation: For every state s_{concr} and all input values in_{abstr}, in_{concr},

if $\mathit{Init}_{concr}(s_{concr})$,
then there exists a state s_{abstr} with $\mathit{Init}_{abstr}(s_{abstr})$ and $R(s_{abstr}, s_{concr})$.

Correctness: For all states $s_{abstr}, s_{concr}, s'_{concr}$, all input values in_{abstr}, in_{concr}, and all output values out_{concr},

if $s_{abstr} \models Op_{abstr}(in)\!\downarrow$ and $R(s_{abstr}, s_{concr})$ and $W(in_{abstr}, in_{concr}, s_{abstr}, s_{concr})$
and $s_{concr}[\![out_{concr} = Op_{concr}(in_{concr})]\!]s'_{concr}$, then there exist s'_{abstr} and out_{abstr} with
$s_{abstr}[\![out_{abstr} = Op_{abstr}(in_{abstr})]\!]s'_{abstr}$ and
$R(s'_{abstr}, s'_{concr})$ or $C(s_{abstr}, s_{concr}, s'_{abstr}, s'_{concr}, out_{abstr}, out_{concr})$.

□

The initialisation condition is the same as for data refinement (Def. 5). The correctness condition for retrenchment is more interesting. It trivially holds if W is *false*, i.e., such states are not further considered. While the within clause excludes states and/or inputs from consideration, the concedes clause C can be used to make the correctness condition valid for state pairs, for which the within clause is true but that are not related via the retrieve relation R.

When gray parts of the condition are removed by defining $W \equiv \mathit{true}$ and $C \equiv \mathit{false}$, the retrenchment condition almost coincides with the standard refinement condition. Then, the only difference is that the input and output values of the concrete and abstract operation may still differ, which is not allowed for refinement. By defining $W \equiv (in_{abstr} = in_{concr})$ we can ensure that the input values are identical but there is no way to express that the output values have to be the same since the concedes clause is disjunctively connected. In order to fix this we supplement the retrieve relation with an *output* clause O.

Definition 8. (Output Retrenchment) Let $\mathcal{A}$ and $\mathcal{C}$ be ADTs as in Def. 7.

The ADT $\mathcal{C}$ is an *output retrenchment* of the ADT $\mathcal{A}$ via

- a *retrieve* relation $R \subseteq S_{abstr} \times S_{concr}$,
- a *within* relation $W \subseteq \mathit{Input}_{abstr} \times \mathit{Input}_{concr} \times S_{abstr} \times S_{concr}$,
- a *concedes* relation $C \subseteq S_{abstr} \times S_{concr} \times S_{abstr} \times S_{concr} \times \mathit{Output}_{abstr} \times \mathit{Output}_{concr}$, and
- an *output* relation $O \subseteq \mathit{Output}_{abstr} \times \mathit{Output}_{concr}$,

iff the following holds:

Initialisation: As in Def. 7.

Correctness: For all states $s_{abstr}, s_{concr}, s'_{concr}$, all input values in_{abstr}, in_{concr}, and all output values out_{concr},

$$\begin{array}{l} \text{if } s_{abstr} \models Op_{abstr}(in)\downarrow \text{ and } R(s_{abstr}, s_{concr}) \text{ and } W(in_{abstr}, in_{concr}, s_{abstr}, s_{concr}) \text{ and} \\ s_{concr}[\![out_{concr} = Op_{concr}(in_{concr})]\!]s'_{concr} \text{ , then there exists } s'_{abstr}, out_{abstr} \text{ with} \\ \quad s_{abstr}[\![out_{abstr} = Op_{abstr}(in_{abstr})]\!]s'_{abstr} \text{ and} \\ \quad (R(s'_{abstr}, s'_{concr}) \text{ and } O(out_{abstr}, out_{concr})) \text{ or } C(s_{abstr}, s_{concr}, s'_{abstr}, s'_{concr}, out_{abstr}, out_{concr}) \text{ .} \end{array}$$

□

Now, in addition to the retrieve relation, the output relation has to hold as well. This generalisation of retrenchment is called output retrenchment [BJ04].[2]

2.4. Operation Retrenchment

We now define a special case of retrenchment, called operation retrenchment, where—like in operation refinement—abstract and concrete operations act on the same state space. Thus, a retrieve relation relating abstract and concrete states is not required and we get a simpler definition:

Definition 9. (Operation Retrenchment) An operation

$$Op_{concr} \subseteq S \times Input_{concr} \times S \times Output_{concr}$$

is an *operation retrenchment* of an operation

$$Op_{abstr} \subseteq S \times Input_{abstr} \times S \times Output_{abstr}$$

(over the same state space S) via

- a within relation $W \subseteq Input_{abstr} \times Input_{concr} \times S$,
- a concedes relation $C \subseteq S \times S \times Output_{abstr} \times Output_{concr}$,
- and an output relation $O \subseteq Output_{abstr} \times Output_{concr}$,

iff, for all states s, s', all input values in_{abstr}, in_{concr}, and all output values out_{concr},

$$\begin{array}{l} \text{if } W(in_{abstr}, in_{concr}, s) \text{ and } s[\![out_{concr} = Op_{concr}(in_{concr})]\!]s' \text{ ,} \\ \text{then there exists an output value } out_{abstr} \text{ with} \\ \quad s[\![out_{abstr} = Op_{abstr}(in_{abstr})]\!]s' \text{ and} \\ \quad O(out_{abstr}, out_{concr}) \text{ or } C(s, s', out_{abstr}, out_{concr}) \text{ .} \end{array}$$

□

2.5. Correctness in the Presence of Retrenchment

In case of a (correct) refinement relation between an abstract program and a concrete program the question of correctness with respect to a specification is easy to answer: from the correctness of the refinement relation and the correctness of the abstract program immediately follows the correctness of the concrete program. In case of retrenchment, however, the situation is a bit more involved.

Theorem 2. (Correctness w.r.t. Retrenchment) Let Op_{abstr}, Op_{concr} be operations, and let $(Pre, Post)$ be a specification.

If

1. Op_{abstr} is correct with respect to $(Pre, Post)$ (Def. 4), and
2. Op_{concr} is an operation retrenchment of Op_{abstr} (Def. 9) via
   - a within relation $W \subseteq Input_{abstr} \times Input_{concr} \times S$,
   - a concedes relation $C \subseteq S \times S \times Output_{abstr} \times Output_{concr}$, and

[2] In the original paper by Banach and Jeske [BJ04], the output relation depends on the same arguments as the concedes clause, but for our purposes the abstract and concrete outputs out_{abstr} and out_{concr} are sufficient.

- an output relation $O \subseteq \mathit{Output}_{abstr} \times \mathit{Output}_{concr}$,

3. for all input values in_{concr} and states s,

   (a) in case that $(in_{abstr}, in_{concr}, s) \in W$ for some input value in_{abstr} *(cases where the within clause holds)*,

   if $(s, in_{concr}) \in \mathit{Pre}$,
   then
   $O(out_{abstr}, out_{concr})$ or $C(s, s', out_{abstr}, out_{concr})$
   implies
   $in_{abstr} = in_{concr}$ and $out_{abstr} = out_{concr}$
   or
   $(s, in_{concr}, s', out_{concr}) \in \mathit{Post}$
   for all states s' and output values out_{concr} with $s[\![out_{concr} = Op(in_{concr})]\!]s'$,

   (b) in case that $(in_{abstr}, in_{concr}, s) \notin W$ for all input values in_{abstr} *(cases excluded by the within clause)*,

   if $(s, in_{concr}) \in \mathit{Pre}$,
   then $(s, in_{concr}, s', out_{concr}) \in \mathit{Post}$
   for all states s' and output values out_{concr} with $s[\![out_{concr} = Op(in_{concr})]\!]s'$,

then Op_{concr} is correct with respect to $(\mathit{Pre}, \mathit{Post})$. □

Proof. Let s be a state and in_{concr} be an input with $(s, in_{concr}) \in \mathit{Pre}$.

We have to show that $(s, in_{concr}, s', out_{concr}) \in \mathit{Post}$ holds for all states s' and output values out_{concr} with $s[\![out_{concr} = Op_{concr}(in_{concr})]\!]s'$.

Case 1: In case that $(in_{abstr}, in_{concr}, s) \notin W$ for all input values in_{abstr}, Condition (3.b) applies, and $(s, in_{concr}, s', out_{concr}) \in \mathit{Post}$ follows immediately.

Case 2: In case that $(in_{abstr}, in_{concr}, s) \in W$ for some input value in_{abstr}, we can conclude that at least one of the clauses $O(out_{abstr}, out_{concr})$ and $C(s, s', out_{abstr}, out_{concr})$ holds from the assumption that Op_{concr} is a retrenchment of Op_{abstr}. That implies $in_{abstr} = in_{concr}$ and $out_{abstr} = out_{concr}$ or $(s, in_{concr}, s', out_{concr}) \in \mathit{Post}$ by Condition (3.a). In the latter case we are done. In the former case, if $in_{abstr} = in_{concr}$ and $out_{abstr} = out_{concr}$, we use the assumption $(s, in_{concr}) \in \mathit{Pre}$, i.e., the precondition holds for the concrete operation. Since the input values in_{abstr}, in_{concr} are equal, that implies $(s, in_{abstr}) \in \mathit{Pre}$. By Condition (1), we know that the abstract operation satisfies the specification. Thus, $(s, in_{abstr}, s', out_{abstr}) \in \mathit{Post}$, and, using the identity of the output values out_{abstr}, out_{concr}, we get $(s, in_{concr}, s', out_{concr}) \in \mathit{Post}$. □

As compared to refinement, when retrenchment is used to establish correctness of the concrete operation Op_{concr}, the additional Condition (3) in Theorem 2 has to be proved.

Note that, although Condition (3.a), which handles cases for which the within clause W is true, looks more complicated than Condition (3.b), it is actually the “harmless” one. In particular, if the definitions

$$\begin{array}{lll} O(out_{abstr}, out_{concr}) & \equiv & in_{abstr} = in_{concr} \wedge out_{abstr} = out_{concr} \\ C(\ldots) & \equiv & \mathit{false} \end{array}$$

for the output and the concedes clauses are used, Condition (3.a) is trivially true for all cases where it applies. Even if other definitions for O and C are used, the particular postcondition Post has rarely to be considered. On the other hand, checking Condition (3.b) always involves the particular postcondition and, thus, the particular specification. To conclude, if O and C are well chosen, Condition (3.a) it trivially true, or at least can be proven once and for all, whereas Condition (3.b) has to be proven for each particular specification.

3. Retrenching Integers

In this paper, we are not concerned with refinement steps within the specification, such as the refinement of a Z or OCL specification into a more precise Z respectively OCL specification, because as long as the language remains the same, at least the *built-in* integer types remain the same.

Instead we assume that we have a refined specification that is supposed to be implemented using the Java programming language. We split this step into two:

1. We assume that the implementation language offers the same infinite types as the specification language, i.e., we assume that the Java type `int` exactly corresponds to the mathematical integers $\mathbb{Z}$. We prove that, using this assumption, the implementation correctly refines the specification. This can, for example, be proven with the KeY system (see Sect. 4).
2. Now, we consider the relationship between the different integer types. The assumption from (1.) is dropped, i.e., instead of a virtual Java program written in a language with infinite integer types we consider the real Java language. Note, that this second step is not a correct refinement but only a retrenchment. Hence, correctness of the real Java implementation does not automatically follow from the correctness of the virtual Java implementation. Rather, additional retrenchment conditions have to be proven, which can also be done using the KeY system.

This break down of the step from specification to code is justified by the fact that programmers may not be aware of the finiteness of the Java types, and even those that are aware of the problem tend to ignore it. Thus, many programmers work with the misconception of infinite integers. For most applications this is not crucial since the range of the Java types is big enough most of the time. But in critical applications this issue has to be handled correctly and cannot be ignored.

In this paper, we focus on the second step, which in the following is cast into the retrenchment framework. We present and compare two possible retrenchments that differ in the definition of the within and concedes clauses. They, thus, differ in the handling of situations where operations on the finite Java integers have a different behaviour than the corresponding operations on the infinite integers. The first retrenchment, which we call $Retrench_{JLS}$, exactly reflects the Java semantics defined in the Java Language Specification (JLS) [GJSB00]. It has a non-trivial concedes clause C, whereas the within clause W only requires the abstract and the concrete input to be equal. In contrast, the second possibility, which we call $Retrench_{KeY}$, has a non-trivial within clause, whereas the concedes clause only requires the abstract and the concrete output to be equal. Both retrenchments are implemented in the KeY system (see Sect. 4) for comparison reasons but, as we will explain in Sect. 3.4, the KeY approach strongly suggests using $Retrench_{KeY}$. Since in both approaches either the within or concedes clause is the same as for refinement, i.e. abstract and concrete input respectively output have to be equal, the two approaches are in some sense two extremes of retrenchments but by no means the only possibilities. One could also think of retrenchments that have both non-trivial within and concedes clauses.

First, we have to define the family of abstract operations that are retrenched in the following sections. They exactly correspond to the usual mathematical functions on $\mathbb{Z}$, namely addition, subtraction, multiplication, division, and modulo.

Definition 10. Let S be an arbitrary set of states. Then, for $\circ \in \{+, -, *, /, \%\}$, the (abstract) operation

$$Op_{abstr,\circ} \subseteq S \times (\mathbb{Z} \times \mathbb{Z}) \times S \times \mathbb{Z}$$

is defined, for all states s, s', input values $\langle in_1, in_2 \rangle$ and output values out, by

$$s[\![out = Op_{abstr,\circ}(\langle in_1, in_2 \rangle)]\!]s' \quad \text{iff} \quad s = s' \text{ and } out = in_1 \circ in_2 \ .$$

□

3.1. Retrenchment by Weakening the Postcondition

Arithmetical operations in Java cause a so-called overflow if and only if the result of an operation would exceed the range of the type. If overflow occurs, the result is calculated modulo the size of the type. Since `int` has a range from $MIN_{int} = -2147483648$ to $MAX_{int} = 2147483647$, we first have to normalise the result before applying the modulo function. Undoing the normalisation thereafter yields the correct result according to the Java semantics.

Definition 11. The function $jmod : \mathbb{Z} \to \mathbb{Z}$ is defined as

$$jmod(x) \ = \ (x - MIN_{int}) \ mod \ (-2 * MIN_{int}) + MIN_{int} \ .$$

□

Definition 12. For $\circ \in \{+,-,*,/,\%\}$, the (concrete) operation

$$Op^{Java}_{concr,\circ} \subseteq S \times (\mathbb{Z} \times \mathbb{Z}) \times S \times \mathbb{Z}$$

is defined, for all states s, s', input values $\langle in_1, in_2\rangle$ and output values out, by

$$s[\![out = Op^{Java}_{concr,\circ}(\langle in_1, in_2\rangle)]\!]s' \quad \text{iff} \quad s = s' \text{ and } out = jmod(in_1 \circ in_2) \ .$$

□

The following theorem provides the within, the concedes, and the output relation for which the operations $Op^{Java}_{concr,\circ}$ are retrenchments of $Op_{abstr,\circ}$ ($\circ \in \{+,-,*,/,\%\}$).

Theorem 3. For every $\circ \in \{+,-,*,/,\%\}$, the operation $Op^{Java}_{concr,\circ}$ is an operation retrenchment of $Op_{abstr,\circ}$ via the relations defined by:

- $W(in_{abstr}, in_{concr}, s)$ iff $in_{abstr} = in_{concr}$,
- $C(s, s', out_{abstr}, out_{concr})$ iff $out_{concr} = jmod(out_{abstr})$ and $out_{abstr} \neq out_{concr}$,[3]
- $O(out_{abstr}, out_{concr})$ iff $out_{abstr} = out_{concr}$.

□

Proof. Given states s, s', input values $in_{abstr} \in \mathbb{Z} \times \mathbb{Z}, in_{concr} \in \mathbb{Z} \times \mathbb{Z}$, and an output value $out_{concr} \in \mathbb{Z}$, we have to show that, under the assumptions

- $in_{abstr} = in_{concr}$ and
- $s[\![out_{concr} = Op^{Java}_{concr}(in_{concr})]\!]s'$,

there exists an output value out_{abstr} with

- $s[\![out_{abstr} = Op_{abstr}(in_{abstr})]\!]s'$ and
- $out_{abstr} = out_{concr}$ or
  $out_{concr} = jmod(out_{abstr})$ and $out_{abstr} \neq out_{concr}$.

But this follows immediately from Def. 10 and 12. □

3.2. Incidental Correctness

As explained in Sect. 2.5, retrenchment is—in contrast to refinement—not correctness preserving. Nevertheless, correctness of the concrete program can be established by proving additional conditions. In certain cases, in particular if a non-trivial concedes clause is used, that may lead to what we call "incidental correctness", a term that is explained in the following.

3.2.1. Example

Assume that a specification for an operation *addTwo* is given that consists of the precondition $even(x)$ and the postcondition $even(result)$, where *result* denotes the output of the operations. The Java implementation of *addTwo* is as follows:

```
int addTwo(int x) {
  return x+2;
}
```

Considering the first step described in Sect. 2.1, i.e., assuming that the Java type coincides with the mathematical integers, it is obvious that the implementation satisfies the specification.

[3] The condition $out_{abstr} \neq out_{concr}$ could be omitted but has the advantage of making the concedes clause C and the output clause O disjoint.

In the second step, we handle the retrenchment where the infinite integer type is replaced by the finite JAVA type `int`. Obviously, for every input $in_{concr} \in \texttt{int}$ there is an input $in_{abstr} \in \mathbb{Z}$ with $in_{abstr} = in_{concr}$, i.e., an abstract value for which W holds. Thus, we can forget about the proof obligation from Condition (3.b) in Theorem 2, and we only have to consider Condition (3.a), which in the example reduces to:

if $even(x)$,
then
  $x +_{\mathbb{Z}} 2 = x +_{\texttt{int}} 2$ or $x +_{\texttt{int}} 2 = jmod(x +_{\mathbb{Z}} 2)$
implies
  $x +_{\mathbb{Z}} 2 = x +_{\texttt{int}} 2$ or $even(x +_{\texttt{int}} 2)$.

Let us consider the premiss of the implication. The first disjunct constitutes the "normal" case where the concrete operation yields the same result as the abstract operation. It obviously implies the right-hand side of the implication. The second disjunct is the concedes clause of the retrenchment, which says that the result of the concrete operation is equal to the result of the abstract operation modulo $-2 * MIN_{int}$, i.e., this is the case where the concrete operations does not correctly simulate the abstract one. Incidentally, however, in our example $-2 * MIN_{int}$ is even and thus also $(x + 2)\ mod\ (-2 * MIN_{int})$ is even under the precondition that x is even. This means that, even if the concrete operation has a different behaviour than the abstract one the postcondition is still satisfied, i.e. the implementation is correct.

3.2.2. The Danger of Incidental Correctness

In general, we call a program *incidentally correct*, if its correctness has been established using retrenchment Theorem 2 via a necessarily non-trivial concedes clause C (i.e., the theorem does not hold any more if C is replaced by *false*).

Correctness in this case may be called incidental since "luckily" the postcondition holds even if the concrete type yields a result different from the abstract result. A user who is not aware of the "luck" involved may think that the same automatically holds true for other postconditions, which it does not.

Programmers tend to use retrenched types as if they were refined types, i.e., the fact that there is a non-trivial concedes clause is ignored. And in fact, the concedes clause often does not play a role since the concrete program solely works on the part of the domain where the concedes clause is not required (most JAVA programs do not exceed (overflow) the range of the type `int`). However, for critical applications we cannot trust our luck and hope that a program does not "make use" of the concedes clause. If it does, there are two possibilities. First, the postcondition may not follow from the concedes clause. Then the program is in fact incorrect. Second, the concedes clause may imply the postcondition, in which case the program is correct. Still, we argue that the second case may cause problems in an ongoing development process. The reason is that the program can run into situations where the concrete type has a different behaviour than the abstract one, though the specification still holds in these exceptional cases. Since the program behaves correctly it may remain hidden from the programmer that the program runs into exceptional situations. Only an inspection of the correctness proof would reveal the fact the concedes clause is actually used. If the developer does not do that (the proof may be constructed automatically or by someone else), the true behaviour may diverge from the developer's intuition and understanding of the program. This is dangerous in an ongoing software project, where programs and even specifications are often modified. Then, a wrong understanding of the internal program behaviour and the fact that the particular concedes clause may not work for other postconditions easily leads to errors that are hard to find, precisely because the program behaviour is not understood.

3.3. Retrenchment by Strengthening the Precondition

The operations $Op^{KeY}_{concr,\circ}$ ($\circ \in \{+, -, *, /, \%\}$) that we define for the second retrenchment differ from the abstract operations only on those parts of the input domain $\mathbb{Z} \times \mathbb{Z}$ where the result of the operation would exceed the bounds MIN_{int} or MAX_{int}. For these cases $Op^{KeY}_{concr,\circ}$ does not terminate.

Definition 13. The predicate $Range \subseteq \mathbb{Z}$ is defined by

$$Range(x) \quad \text{iff} \quad MIN_{int} \leq x \leq MAX_{int} \ .$$

□

Definition 14. For $\circ \in \{+,-,*,/,\%\}$, the concrete operation

$$Op^{KeY}_{concr,\circ} \subseteq S \times (\mathbb{Z} \times \mathbb{Z}) \times S \times \mathbb{Z}$$

is defined, for all states s, s', input values $\langle in_1, in_2 \rangle$, and output values out, by

$$s[\![out = Op^{KeY}_{concr,\circ}(\langle in_1, in_2\rangle)]\!]s' \quad \text{iff} \quad \begin{array}{ll} \text{(i)} & s = s' \ , \\ \text{(ii)} & out = in_1 \circ in_2 \ \text{, and} \\ \text{(iii)} & \text{if } Range(in_1) \text{ and } Range(in_2) \text{ then } Range(out) \ . \end{array}$$

□

Theorem 4. For every $\circ \in \{+,-,*,/,\%\}$, the operation $Op^{KeY}_{concr,\circ}$ is an operation retrenchment of $Op_{abstr,\circ}$ via the relations defined by:

- $W(in_{abstr}, in_{concr}, s)$ iff (i) $in_{abstr} = in_{concr}$ and (ii) if $Range(in_{concr,1})$ and $Range(in_{concr,2})$ then $Range(in_{concr,1} \circ in_{concr,2})$,
- $C(s, s, out_{abstr}, out_{concr}) \equiv false$,
- $O(out_{abstr}, out_{concr})$ iff $out_{abstr} = out_{concr}$.

□

Proof. Given states $s, s' \in S$, input values $\langle in_{abstr,1}, in_{abstr,2}\rangle, \langle in_{concr,1}, in_{concr,2}\rangle \in \mathbb{Z} \times \mathbb{Z}$, and an output value $out_{concr} \in \mathbb{Z}$, we have to show, under the assumptions

- $s[\![out_{concr} = Op^{KeY}_{concr}(in_{concr})]\!]s'$,
- $in_{abstr} = in_{concr}$, and
- if $Range(in_{concr,1})$ and $Range(in_{concr,2})$ then $Range(in_{concr,1} \circ in_{concr,2})$,

that there exists an output value out_{abstr} with

- $s[\![out_{abstr} = Op_{abstr}(in_{abstr})]\!]s'$ and
- $out_{abstr} = out_{concr}$

which follows immediately from Def. 10 and 14. □

For the KeY system, we make use of the following corollary of Theorem 4 stating that a program where the abstract operations $Op_{abstr,\circ}$ (see Def. 10) are replaced with the retrenched operations $Op^{KeY}_{concr,\circ}$ (see Def. 12) is correct if the program is correct before and the within clause W of the retrenchment always holds when the concrete operation is applied.

Corollary 1. Let p_{abstr} be a program using the abstract operations $Op_{abstr,\circ}$, and let p_{concr} be a program that is the result of replacing $Op_{abstr,\circ}$ in p_{abstr} with the concrete operations $Op^{KeY}_{concr,\circ}$ ($\circ \in \{+,-,*,/,\%\}$). Further, let $(Pre, Post)$ be a specification.

If

1. p_{abstr} satisfies $(Pre, Post)$,
2. For all occurrences of the abstract operations in p_{abstr} with input $\langle in_{abstr,1}, in_{abstr,2}\rangle$, the respective occurrences of concrete operations in p_{concr} are invoked with inputs $\langle in_{concr,1}, in_{concr,2}\rangle$ such that

$$\text{if } Range(in_{concr,1}) \text{ and } Range(in_{concr,2}) \text{ then } Range(in_{concr,1} \circ in_{concr,2}) \ ,$$

then p_{concr} satisfies $(Pre, Post)$. □

When the program verification calculus implemented in the KeY system (see Sect. 4.2) is used to prove that a program p_{abstr} satisfies a specification $(Pre, Post)$ (Condition (1) in Corollary 1), it automatically generates proof goals for Condition (2) as well. Thus, when the proof succeeds, Corollary 1 implies that not only does p_{abstr} satisfy the specification but so does p_{concr}.

Proving p_{concr} to be correct, however, is not really the end of the story. What we actually have to establish is the correctness of the JAVA implementation p_{JAVA} with operations $Op^{Java}_{concr,\circ}$. Correctness of p_{JAVA} does not automatically follow from the correctness of p_{concr} since the operations $Op^{Java}_{concr,\circ}$ are not correct refinements of $Op^{KeY}_{concr,\circ}$ making another retrenchment step necessary.

Fortunately, the operations $Op^{KeY}_{concr,\circ}$ are designed in such a way as to make this final retrenchment step quite simple. The operations $Op^{Java}_{concr,\circ}$ are operation retrenchments of the operations $Op^{KeY}_{concr,\circ}$ via the within condition

$$W(in_{abstr}, in_{concr}, s) \text{ iff } Range(in_{concr,1}) \text{ and } Range(in_{concr,2}) \ .$$

For Java programs that within condition W is trivially satisfied because a Java variable and, thus, the input to a Java operation can never have a value outside the range of its type. It is, therefore, not necessary to generate proof obligations for the above within clause (i.e., this particular retrenchment behaves similar to refinement).

3.4. Comparison

We argue that the operations $Op^{KeY}_{concr,\circ}$ and retrenchment $Retrench_{KeY}$ are more suited for program verification than the operations $Op^{Java}_{concr,\circ}$ and $Retrench_{JLS}$, for the following reasons.

- Retrenchment $Retrench_{KeY}$ has, in contrast to $Retrench_{JLS}$, a trivial concedes clause $W \equiv \mathit{false}$ and thus prevents incidentally correct programs, which are a source of error in an ongoing software development process (see Sect. 3.2).
- Using $Retrench_{JLS}$ introduces the modulo function into arithmetical terms, which makes proving more complicated and unintuitive. Our experience shows that many proof goals involving integer arithmetics (that remain after the rules of our program verification 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 [SBD02] and the Simplify tool, which is part of ESC/Java [ESC]. Both do *not* work for modulo arithmetics.

4. Implementation

In this section we describe how the retrenchment $Retrench_{KeY}$ described in Sect. 3.3 is implemented in the KeY system.

4.1. Background

The work reported in this paper has been carried out as part of the KeY project [ABB$^+$05, ABB$^+$00] (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) [Obj99] has been widely accepted as 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) since it is a relevant language that is used in practice and that is, due to its design, relatively well accessible to program verification (compared to other language like, e.g., C++).

We use an instance of dynamic logic (DL) [Har84, HKT00, KT90, Pra77]—which can be seen as an extension of Hoare logic—as the logical basis of the KeY system's software verification component. Deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of Java. DL is used in the software verification systems KIV [BRS$^+$00] and VSE [HLS$^+$96] for (artificial) imperative programming languages. More recently, the KIV system supports also a fragment of the Java language [Ste01]. In both systems, DL was successfully applied to verify software systems of considerable size.

DL can be seen as a modal logic with a modality $\langle p \rangle$ for every program p (we allow p to be any sequence of legal Java Card 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 such states (worlds) because the programs can be non-deterministic; but here, since JAVA CARD 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 $\langle p\rangle\psi$ expresses that the program p terminates in a state in which ψ holds. A formula $\phi \rightarrow \langle p\rangle\psi$ is valid if for every state s satisfying pre-condition ϕ a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds. The formula $\phi \rightarrow [p]\psi$ expresses the same, except that termination of p is not required, i.e., ψ only has to hold *if* p terminates.

Thus, the formula $\phi \rightarrow [p]\psi$ is equivalent to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators: In Hoare logic, the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. DL allows to involve programs in the descriptions ϕ respectively ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Also, all JAVA constructs are available in our DL for the description of states (including `while` loops and recursion). It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type; instead DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows to concentrate on the relevant properties of a state.

4.2. Sequent Calculus for Integer Retrenchment

4.2.1. Overview

The KeY system's deduction component uses the program logic JavaDL, which is a version of Dynamic Logic modified to handle JAVA CARD programs [Bec01]. We have extended and adapted that calculus to implement the approach to handling integer arithmetic based on the retrenchments presented in Sect. 3.1 and 3.3. Although we strongly recommend using $Retrench_{KeY}$ and not $Retrench_{JLS}$ (see the comparison in Sect. 3.4), the user can configure the KeY prover to use any of the two possibilities. We concentrate on $Retrench_{KeY}$ in the following.

Here, we cannot list all rules of the adapted calculus (they can be found in [Sch02]). To illustrate how the calculus works, we present some typical rules for expressions of type `int` representing the two different rule types: program transformation rules to evaluate compound JAVA expressions (Sect. 4.2.4) and rules to symbolically execute simple JAVA expressions (Sect. 4.2.5). Similar rules exist for all arithmetical operators of the JAVA types `byte`, `short`, and `long`.

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

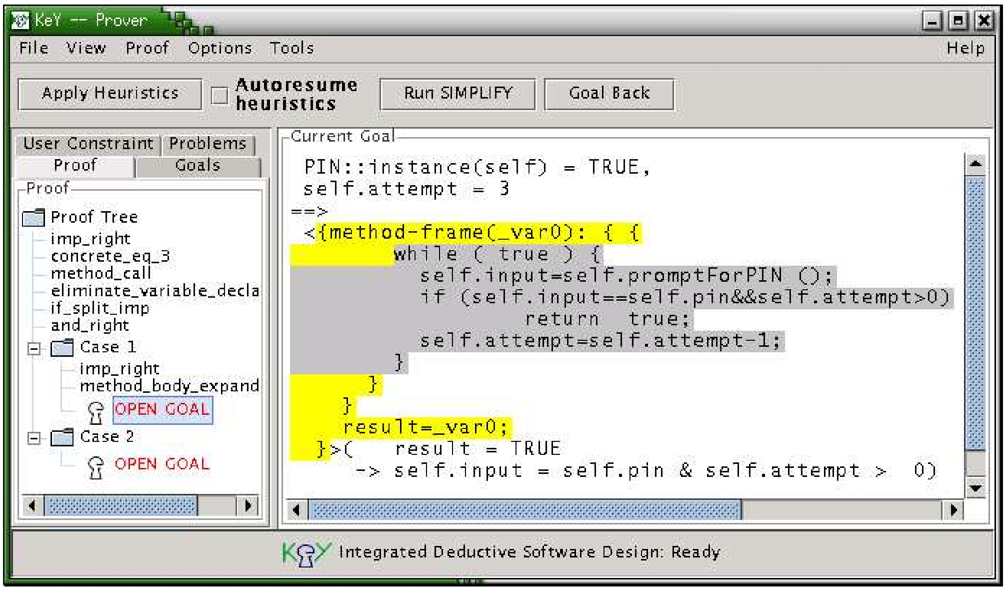
$$(\forall x_1) \cdots (\forall x_k)((\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n)) \;\;,$$

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

4.2.2. 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, *se* 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 rules of our calculus operate on the first *executable* statement p of a program $\pi\, p\, \omega$. The non-active prefix π e.g. consists of opening braces "`{`" and beginnings "`try{`" of `try-catch-finally` blocks. The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the first active statement. E.g., if a rule is applied to the JAVA block " `try{ i=0; j=0; }finally{ k=0; }` ", operating



```
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.

on its first active statement "`i=0;`", then π is " `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 Fig. 1.

4.2.3. State Updates

We allow *updates* of the form $\{x := t\}$ respectively $\{o.a := t\}$ 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 $\langle\texttt{x = t;}\rangle\phi$.

4.2.4. Program Transformation Rules

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

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

T is the (declared) type of *var*. The explicit type cast is necessary to preserve the semantics since the arguments of `+` are internally cast to `int` or `long` which is not the case for the postfix increment operator `++`.

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*).

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

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.

4.2.5. Symbolic Execution Rules

The Rule for Subtraction. This rule symbolically executes a subtraction.

$$\frac{\begin{array}{l}\Gamma,\ \mathit{Range}(se_1) \wedge \mathit{Range}(se_2)\ \vdash\ \mathit{Range}(se_1 - se_2),\ \Delta \\ \Gamma\ \vdash\ \{\mathit{var} := se_1 - se_2\}\langle \pi\ \omega\rangle\phi,\ \Delta\end{array}}{\Gamma\ \vdash\ \langle \pi\ \mathit{var}\ \texttt{=}\ se_1\ \texttt{-}\ se_2\texttt{;}\ \omega\rangle\phi,\ \Delta} \qquad \text{(R3)}$$

The first premiss establishes the within clause W of the retrenchment $\mathit{Retrench}_{KeY}$ (see Theorem 4). If both arguments are within MIN_{int} and MAX_{int}, then the result must be within that range as well (no overflow occurs).

In the second premiss the JAVA statement is symbolically executed, i.e. the statement disappears from the program and is translated into a state update.

Note, that the rule contains two different symbols for subtraction with different semantics: The symbol "`-`" denotes the JAVA operation. It occurs in the conclusion. The symbol "$-$", which occurs in the two premisses, represents the abstract subtraction operation on $\mathbb{Z}$.

4.2.6. Improvements

The calculus presented in Sect. 4.2 generates additional proof obligations for symbolic execution of arithmetical operations, which ensure that the within clause W holds, i.e. that the range where the JAVA integers correctly simulate the mathematical integers is not exceeded. In this section we identify situations where such an additional proof obligation is not necessary since it can be automatically shown a priori that the range is not exceeded. In addition, proof re-use can be used to split a proof into two steps as described in Sect. 3 without duplicating work.

Static Analysis In JAVA there is no polymorphism for primitive types like `int`. The type of expressions can be determined statically. This can be exploited to avoid checking bounds in certain situations. For example, the JAVA expressions `i+j` and `i*j` cannot exceed the bounds MIN_{int} or MAX_{int} if `a`, `b` are of type `short` since the whole expression is of type `int`. Also, constant expressions can be evaluated a priori by static analysis.

Proof Re-Use In Sect. 2.1 we described how the step from specification to code can be split into two parts: in a first step, we assume that the JAVA integers are infinite and prove the correctness relative to that semantics. Then, in the second step, we use the real JAVA types and show that they are only used on parts of the domain where the within clause W holds.

KeY offers two different integer semantics corresponding to the two steps described above. For the first step we can choose the mathematical semantics and prove the correctness relative to that semantics. If that succeeds, we then can change the integer semantics to the one from $\mathit{Retrench}_{KeY}$ and repeat the proof. However, we do not have to redo all the work. The only difference between the two proofs will be that for each arithmetical operation an additional proof obligation is generated when using $\mathit{Retrench}_{KeY}$. The KeY prover offers a facility for proof re-use [BK04], which is very helpful in this case. Applying proof re-use, only the additional goals remain. The other goals are closed automatically by the re-use mechanism since they are identical to the first proof. The advantage of splitting the two steps is that, if already the first step fails, the second is not considered before the problem has been fixed.

5. Example

The following extended example illustrates our approach for handling the retrenchment from the infinite integer type $\mathbb{Z}$ to the finite JAVA programming language type `int`. 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 gets his credit card back. Otherwise, if the PIN is incorrect, two more attempts are allowed 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.

```
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 int 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. Two possible implementations of method `pinCheck`.

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).

5.1. Formal Specification and Implementation

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`.

The above formal specification is not complete (with respect to the informal specification):[4] 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.

First, we consider the possible implementation of the method `pinCheck` shown on the left in Fig. 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.

5.2. Verifying the Implementation

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

[4] 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.

$$\texttt{attempt} = 3 \;\vdash\; \langle p \rangle(\texttt{result} = \mathit{true} \rightarrow \texttt{input} = \texttt{pin} \wedge \texttt{attempt} > 0) \tag{S1}$$

Fig. 1 shows this sequent after "unpacking" the method body of `pinCheck` in the KeY prover.

With this example we also want to show the advantage of using retrenchment $\mathit{Retrench}_{KeY}$ and not $\mathit{Retrench}_{JLS}$. Both retrenchments are implemented and the user of the KeY system can configure which of the two retrenchments he or she wants to use. First, we chose $\mathit{Retrench}_{JLS}$ and point out problems that occur when trying to prove the implementation from above. Then, we show how these problems can be avoided following the KeY approach and using the retrenchment $\mathit{Retrench}_{KeY}$.

5.2.1. Verification with $\mathit{Retrench}_{JLS}$

If we use the integer retrenchment $\mathit{Retrench}_{JLS}$ presented in Sect. 3.1, which exactly reflects the Java semantics, then sequent (S1) is derivable. Consequently, the implementation is correct in the sense that it satisfies the formal specification.

But this implementation has an unintended behaviour. Suppose the credit card has been stolen and someone wants to withdraw money without knowing the PIN. Let us assume that there is no other possibility than a brute force attack, i.e. trying all possible PINs. According to the informal specification, after three wrong attempts any further attempt should not be successful any more. But with our implementation at some point the counter `attempt` will overflow and get the positive value MAX_{int}, i.e. in fact the attacker has many attempts to guess the right PIN and eventually to withdraw money.

The main reasons for this unexpected behaviour are the incomplete formal specification and the implementation that is "incidentally" correct (see Sect. 3.2) with respect to the (inadequate) formal specification.

5.2.2. Verification with $\mathit{Retrench}_{KeY}$

Now we use the retrenchment $\mathit{Retrench}_{KeY}$ instead of $\mathit{Retrench}_{JLS}$ and try to derive sequent (S1) again.

We do not show all the proof steps and the corresponding rules that have to be applied. Rather, we concentrate on the crucial point in the proof when it comes to handle the statement "`attempt=attempt-1;`". After applying rule (R3), one of the new goals is the following:

$$\mathit{Range}(\texttt{attempt}),\ \mathit{Range}(1) \;\vdash\; \mathit{Range}(\texttt{attempt} - 1)\ .$$

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 would cause overflow). The sequent does not hold because its left side is true but its right side is false (as $\texttt{attempt} - 1$ is not in valid range). Fig. 3 shows the invalid sequent in the KeY prover where *Range* is already expanded.

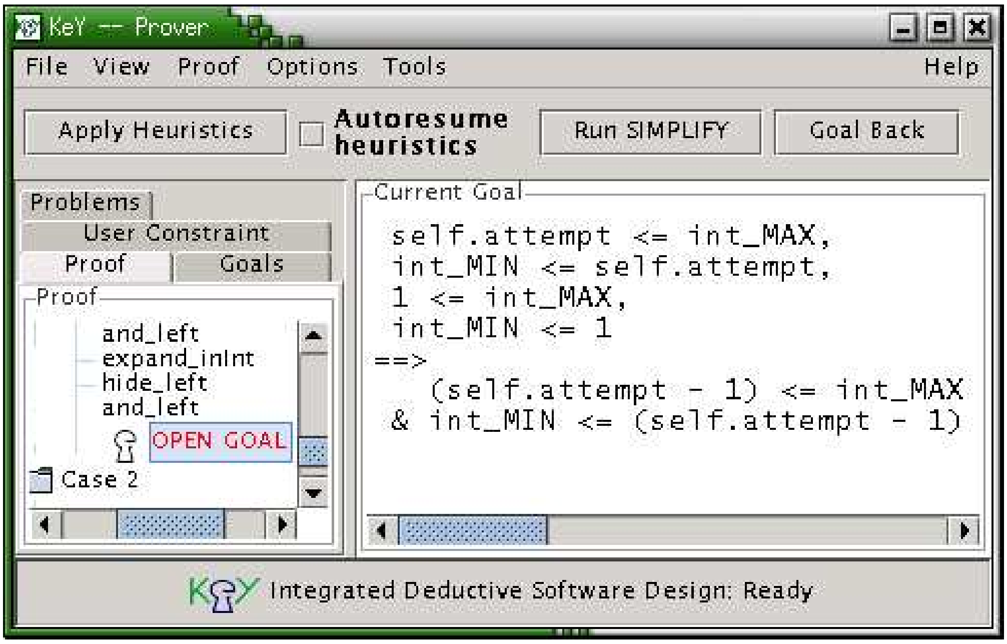
Note, that this error is uncovered by using a semantics for Java integer arithmetic based on retrenchment $\mathit{Retrench}_{KeY}$. If $\mathit{Retrench}_{JLS}$ is used instead this error is not detected.

5.3. Revising the Implementation

Since the proof obligation (S1) is not derivable in our calculus when using $\mathit{Retrench}_{KeY}$, the implementation must be revised for being 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 of Fig. 2. Trying to verify this new implementation with the KeY system leads to the sequent shown in Fig. 4. In contrast to the one shown in Fig. 3, this sequent is valid because of the additional formula $(\texttt{self.attempt}) > 0$ on the left side, which stems from the check added in the revised implementation.

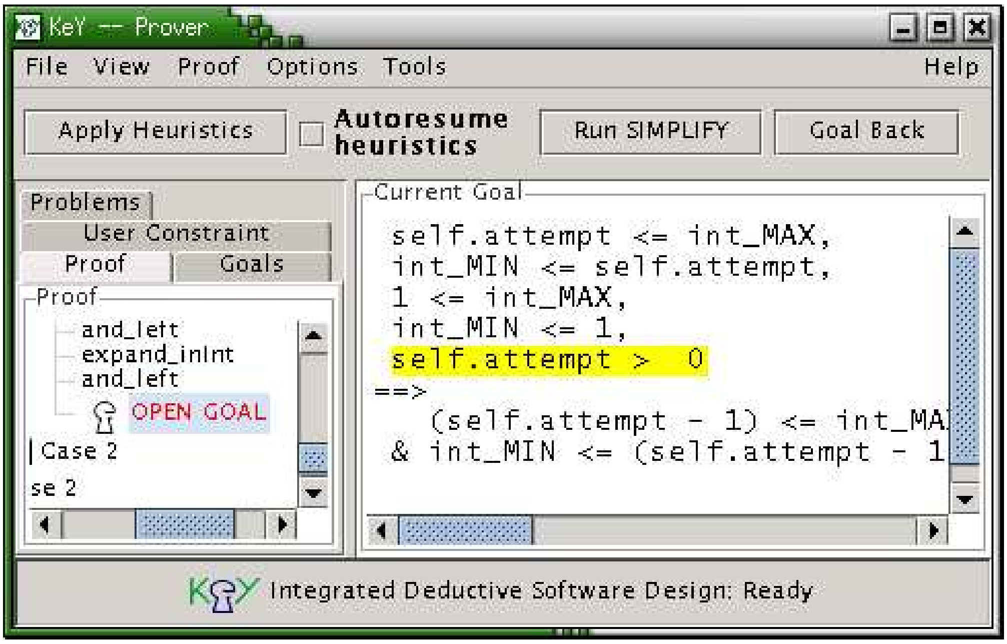
The resulting proof obligation can now be derived in our calculus and, thus, Corollary 1 implies that the revised implementation satisfies the specification. But we also know that this correctness is not incidental, which here means that no overflow occurs when the program is executed on the Java virtual 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



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

Fig. 3. The KeY prover window with an invalid sequent.



```
 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. 4. The KeY prover window with the sequent from Fig. 3 plus the highlighted premiss that makes the sequent valid.

intended by the programmer. Following our approach, and using $Retrench_{KeY}$, 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. If a program cannot be proved correct 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. Related Work

Retrenchment was first mentioned in [BP98] as an answer to the problem that refinement is too restrictive for many practical applications. In [BP99, BJ04] even more general variants of retrenchment are presented, e.g. output retrenchment.

Research on arithmetic in verification focused so far mainly formalising and verifying properties of floating-point arithmetic [Har99, Har00] (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 [ESA96] 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 [Jac03, Ste01].

The verification techniques described in [PHM99, vO01, Hui01] treat Java's integer types as if they were infinite, i.e., the overflow problem is ignored.

Closely related to our approach is Chalin's work [Cha03]. 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.

7. Conclusions

Developing software systems by stepwise refinement is often not easy and sometimes even impossible. In particular, the last step from an already refined specification to code often violates the principles of refinement. One reason is that idealistic data types that are available in specification languages, such as the natural numbers $\mathbb{Z}$ and the real numbers $\mathbb{R}$, are not available in programming languages. Instead, programming language data types have to be used that are not a correct refinement of the abstract specification language types. However, the programming language data types are not completely different from the abstract types—on parts of the domain they even behave as if they were a correct refinement. In this paper we used the mathematical integers $\mathbb{Z}$ on the one hand and the primitive Java type `int` on the other hand to illustrate the problems and to describe our approach to overcome the problem.

The idea of "correctness by construction" using refinement has to be adapted when, e.g., replacing $\mathbb{Z}$ with `int`, and additional proofs become necessary to ensure correctness of the system. We used the retrenchment framework [BP98] to formally describe the non-refinement steps. The advantage of casting non-refinement steps into the retrenchment framework is that it becomes explicit where exactly the refinement conditions are violated and, thus, where correctness cannot be shown once and for all (as it is the case with correct refinement). Instead, we prove the correctness of a program containing retrenchment by individually verifying critical situations. After these proofs have been done, no run-time checks are required. The additional proof obligations are systematically generated from the retrenchment clauses. Case studies showed that most of the additional proof obligations can be discharged automatically by the KeY system [ABB$^+$05, ABB$^+$00] or by external decision procedures like CVC [SBD02] and the Simplify tool, which is part of ESC/Java [ESC].

Acknowledgement

We thank Richard Banach, Michael Poppleton, and Eerke Boiten for fruitful discussions on refinement and retrenchment. We also thank the anonymous referees for useful comments and suggestions.

References

[ABB⁺00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter 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.

[ABB⁺05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[BD98] E.A. Boiten and J. Derrick. IO-Refinement in Z. In A Evans, D Duke, and T Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer, September 1998.

[Bec01] Bernhard 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.

[BJ04] R. Banach and C. Jeske. Output retrenchments, defaults, stronger compositions, feature engineering, 2004. Available at `http://www.cs.man.ac.uk/~banach/Recent.publications.html`.

[BK04] Bernhard Beckert and Vladimir Klebanov. Proof Reuse for Deductive Program Verification. In J. Cuellar and Z. Liu, editors, *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China.* IEEE Press, 2004.

[BP98] R. Banach and M. Poppleton. Retrenchment: An Engineering Variation on Refinement. In Didier Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, LNCS 1393, pages 129–147. Springer, 1998.

[BP99] R. Banach and M. Poppleton. Sharp retrenchment, modulated refinement and punctured simulation. *Formal Aspects of Computing*, 11:498–540, 1999.

[BRS⁺00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer-Verlag, 2000.

[Cha03] Patrice Chalin. Improving JML: For a Safer and More Effective Language. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *Proceedings, FME 2003: Formal Methods, Pisa, Italy*, LNCS 2805, pages 440–461. Springer, 2003.

[DB01] John Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Formal Approaches to Computing and Information Technology. Springer, May 2001.

[ESA96] European Space Agency. Ariane 501 inquiry board report, July 1996. Available at: `http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf`.

[ESC] ESC/Java (Extended Static Checking for Java). `http://research.compaq.com/SRC/esc/`.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.

[Har84] David Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs), Nice, France*, LNCS 1690, pages 113–130. Springer, 1999.

[Har00] John 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.

[HHS86] 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.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[HLS⁺96] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, and Werner Stephan. Deduction in the Verification Support Environment (VSE). In M.-C. Gaudel and J. Woodcock, editors, *Proceedings, International Symposium of Formal Methods Europe (FME), Oxford, UK*, LNCS 1051. Springer, 1996.

[Hui01] Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.

[Jac03] Bart 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.

[KT90] Dexter Kozen and Jerzy Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, Amsterdam, 1990.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[Obj99] Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential java. In S. D. Swierstra, editor, *Proceedings, European Symposium on Programming (ESOP), Amsterdam, The Netherlands*, LNCS 1576, 1999.

[Pra77] V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand 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.

[Sch02] Steffen 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`.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[Ste01] Kurt 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/`.

[vO01] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.