CHALMERS
UNIVERSITY OF TECHNOLOGY

Combining rule- and SMT-based reasoning for verifying floating-point Java programs in KeY

Downloaded from: https://research.chalmers.se, 2024-10-12 02:11 UTC

Citation for the original published paper (version of record):
Abbasi, R., Schiffl, J., Darulova, E. et al (2023). Combining rule- and SMT-based reasoning for verifying floating-point Java programs in KeY. International Journal on Software Tools for Technology Transfer, 25(2): 185-204. http://dx.doi.org/10.1007/s10009-022-00691-x

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

International Journal on Software Tools for Technology Transfer (2023) 25:185–204
https://doi.org/10.1007/s10009-022-00691-x

GENERAL

Special Issue: TACAS 2021

Combining rule- and SMT-based reasoning for verifying floating-point Java programs in KeY

Rosa Abbasi[1] · Jonas Schiffl[2] · Eva Darulova[1,3] · Mattias Ulbrich[2] · Wolfgang Ahrendt[4]

Accepted: 7 November 2022 / Published online: 8 March 2023
© The Author(s) 2023

Abstract
Deductive verification has been successful in verifying interesting properties of real-world programs. One notable gap is the limited support for floating-point reasoning. This is unfortunate, as floating-point arithmetic is particularly unintuitive to reason about due to rounding as well as the presence of the special values infinity and 'Not a Number' (NaN). In this article, we present the first floating-point support in a deductive verification tool for the Java programming language. Our support in the KeY verifier handles floating-point arithmetics, transcendental functions, and potentially rounding-type casts. We achieve this with a combination of delegation to external SMT solvers on the one hand, and KeY-internal, rule-based reasoning on the other hand, exploiting the complementary strengths of both worlds. We evaluate this integration on new benchmarks and show that this approach is powerful enough to prove the absence of floating-point special values—often a prerequisite for correct programs—as well as functional properties, for realistic benchmarks.

Keywords Deductive verification · Floating-point arithmetic · Transcendental functions

1 Introduction

Deductive verification has been successful in providing functional verification for programs written in popular programming languages such as Java [2,21,40,48], Python [27], Rust [4], C [23,53], and Ada [17,49]. Deductive verifiers allow a user to annotate methods in a program with pre- and postconditions, from which verification conditions (VCs) are automatically generated. These are then either proven directly by the verifier itself, or discharged with external tools such as automated Satisfiability Modulo Theories (SMT) solvers or interactive proof assistants.

While deductive verifiers fully implement many sophisticated data representations (including heap data structures, objects, and ownership), support for floating-point numbers remains rather limited—solely Frama-C and SPARK offer automated support for floating-point arithmetic in C and Ada [30]. This state of affairs is at least partially a result of previous limitations in floating-point support in SMT solvers. Consequently, deductive verification has been used for floating-point programs only by experts with considerable manual effort [13,30]. This is unfortunate as it makes deductive verification unavailable for a large number of programs across many domains including embedded systems, machine learning, and scientific computing. With the increasing need for parallelization in code, scientific computing specifically has recently experienced algorithmic challenges for which formal methods may contribute to a solution [8,56].

One of the main challenges of floating-point arithmetic is its unintuitive behavior and the special values that the IEEE

✉ Rosa Abbasi
rosaabbasi@mpi-sws.org

Jonas Schiffl
jonas.schiffl@kit.edu

Eva Darulova
eva@mpi-sws.org

Mattias Ulbrich
ulbrich@kit.edu

Wolfgang Ahrendt
ahrendt@chalmers.se

1 MPI-SWS, Kaiserslautern and Saarbrücken, Saarbrücken, Germany

2 Karlsruhe Institute of Technology, Karlsruhe, Germany

3 Uppsala University, Uppsala , Sweden

4 Chalmers University of Technology, Gothenburg, Sweden

754 standard [38] introduces. For instance, an overflow or a division by zero results in the special value (positive or negative) *infinity*, and not a runtime exception. Similarly, invalid operations like `sqrt(-1.0)` result in a *Not a Number* (NaN) value. These special values are problematic as seemingly straightforward identities do not hold (`x == x` or `x * 0.0 == 0.0`). In addition, every operation on floating-point numbers potentially involves rounding, which compromises familiar rules like associativity and distributivity. Hence, reasoning support for writing correct floating-point programs is indispensable.

Abstract interpretation-based tools can prove the absence of runtime errors and special values [18,42] and bound roundoff errors due to floating-point's finite precision [9,19,24,35,58]. SMT decision procedures [15] or SAT-based model-checking [22,56], on the other hand, can prove intricate properties requiring bit-precise reasoning. However, these techniques and tools largely support only purely floating-point programs or program snippets, or analyze programs only up to a predefined depth of the call stack. General reasoning about real-world object-oriented programs, however, requires support for more than floating-point arithmetic. Many realistic programs mix floating-point types with other primitive types, like integers, including (implicit) type casts and moreover use object-oriented concepts, like the (unbounded) object heap. This necessitates different analyses which need to be integrated with floating-point reasoning.

Handling floating-points in a deductive verifier has unique advantages. First, the deductive verification approach already comes with the infrastructure for reasoning about complex control and data structures (like exception handling and heaps). Second, it allows one to flexibly combine the verifier's symbolic execution reasoning with external decision procedures. Third, depending on the theory support, the verifier or external solver may also generate counterexamples of a property and thus help program debugging—something an abstract interpretation-based approach fundamentally cannot provide.

We report on adding floating-point support to the KeY deductive verifier, providing the first automated deductive floating-point support for the Java programming language. Among others, we verify the absence of the special values infinity and NaN. While those are helpful in particular circumstances (like carefully designed implementations of numeric analysis algorithms), for most applications they indicate an error. Hence, showing their absence is a prerequisite for further (functional) reasoning. Moreover, our extension allows to express and discharge functional properties relating pre- and poststates, including bounds on roundoff errors and bounds on differences between two similar floating-point programs, which we also demonstrate.

We exploit both KeY's symbolic execution and external SMT support. On the one hand, we handle arithmetic operations by relying on a combination of KeY's symbolic execution to handle the heap and SMT-based decision procedures to handle the floating-point part of the VCs. On the other hand, we support transcendental functions via axiomatization in the KeY prover itself.

Transcendental functions such as sine are a common feature in engineering applications, but are not supported by floating-point decision procedures. We explore two ways of supporting them soundly but approximately, by encoding them as axiomatized uninterpreted function symbols once directly in the SMT queries, and once in additional calculus rules in KeY. Our evaluation shows that even though such reasoning is approximate, it is nonetheless sufficient to prove the absence of special values in many interesting programs.

We evaluate KeY's floating-point support on a number of real-world floating-point Java programs, verifying the absence of special values as well as functional correctness, including programs with loops (with the help of loop invariants). Our benchmark set allows us to evaluate recent progress in SMT floating-point support in Z3 [26], CVC4 [6] and MathSAT [20] on yet unseen benchmarks. For instance, we observe that quantifiers are challenging even if they do not affect satisfiability of SMT queries. Our benchmarks are openly available, and we expect our insights to be useful for further solver development.

Contributions In summary, we make the following contributions:

- we implement and evaluate the first automated deductive verification of floating-point Java programs by combining the strength of rule-based and SMT-based deduction;
- we develop novel automated support for reasoning about transcendental functions in a deductive verifier;
- we add support for reasoning about (potentially rounding) type conversions;
- we collect a new set of challenging real-world floating-point benchmarks in Java;[1]
- we compare different SMT solvers for discharging floating-point VCs on this new set of benchmarks.

This article extends a previous conference paper [1] by more floating-point arithmetic background, the presentation of all rules for transcendental functions and `sqrt`, new rules—and accordingly experiments—for (potentially rounding) casts between integers and floats, new experiments on the verification of floating-point loop invariants, new experiments on the sensitivity of SMT solvers to numer-

[1] Available at https://zenodo.org/record/6572961#%23.Yot0hJNBy3I

ous problem modifications, and extended reporting of the experiments presented in [1].

2 Background

2.1 Introduction to KeY

KeY [2] is a platform for deductive verification of Java programs, working at a source code level. The input is a Java program annotated in the Java Modeling Language (JML) [44], encouraging a *Design by Contract* ([45,50]) approach to software development. The user specifies the expected behavior of Java classes with *class invariants* that the program has to maintain at critical points. Methods are specified with *method contracts*, consisting mainly of pre- and postconditions, with the understanding that if the precondition holds when the method is called, the postcondition has to hold after the method returns.

After loading an annotated program, KeY translates it to a formula in Java Dynamic Logic [2] (JavaDL), an instance of Dynamic Logic [36] which enables logical reasoning about Java programs. Logical rules are provided for the translation of programs into first-order logic, and for closing the resulting *goals*, or proof obligations. KeY is semi-interactive in that it allows manual rule application, while also offering powerful built-in automation and macros.

The rules are written in KeY as *taclets*,

calculus rule schemata, implementing rewrite rules. One example taclet that rewrites any expression matching $x + 0$ to x is shown below:

```
find    x + 0
replace x
```

In general, taclets have the following form:

```
find    t_find
replace t_replace
add     φ_add
show    φ_show
```

It consists of a schematic `find` term t_{find} which has to match an expression or formula in the current goal, and a term $t_{replace}$ which replaces the matched expression or formula. New formulas may be introduced as new assumptions onto the proof goal using `add`. If the rule has a side condition φ_{show} that needs to be established, it can be optionally specified using `show` that will open a new proof goal for φ_{show}. One of the clauses for `add` and `replace` may be empty or omitted. The taclet language supports typed meta-variables and heuristics for KeY's automation.

In addition to the application of taclets, KeY also supports the translation of open goals into the common SMT input format SMT-LIB [7] and the calling of an external SMT solver. For specific theories, SMT solvers can be much more efficient than KeY's own, rule-based reasoning, while other goals are more effectively dealt with by KeY rather than SMT solvers. KeY therefore allows to discharge some goals with SMT solvers, and others with KeY's own, rule-based proof engine.

2.2 Floating-point arithmetic in Java

Among the primitive (i.e., non-reference) types of Java, there are two which represent floating-point numbers, namely `float` and `double`. They are associated with the 32-bit and 64-bit format, respectively, as specified in the IEEE 754 Standard for Floating-Point Arithmetic. More precisely, Java implements a subset of that standard. In the following, we summarize some central characteristics of Java floating-point numbers, loosely following Muller et.al. [52]. Most of this is not specific to Java, but more generally applies to IEEE 754. Note that the Java Virtual Machine (JVM) only supports floating-point numbers with base 2, even if the Java language syntax supports base 10 as well, in such a way that parsing and input/output routines translate back and forth between the bases.

Each (base 2) floating-point number x (except the special values $+\infty$, $-\infty$, and NaN, see below) with precision p can be represented as a triplet (s, m, e), such that $x = (-1)^s * m * 2^e$, where $s \in \{0, 1\}$ is the *sign*, m (called *significand*) is a binary fixed-point number with one digit before the radix point and $p - 1$ digits after the radix point (note that $0 \leq m < 2$), and e (*exponent*) is an integer such that $e_{min} \leq e \leq e_{max}$. Java supports two floating-point formats (both in base 2): `float` with $p = 24$, $e_{min} = -126$, $e_{max} = 127$ and `double` with $p = 53$, $e_{min} = -1022$, $e_{max} = 1023$.

Whenever the result of a computation cannot be exactly represented with the given precision, it is rounded. IEEE 754 defines various rounding modes, of which Java only supports *round to nearest, ties to even*. Rounding is exact, as if one would first compute the ideal real number, and round afterwards. Note that rounding may even occur when the exact mathematical result of a computation corresponds to an integer number. The single precision type `float` has a significand of 24 bits and can therefore not exactly represent all integers of the 32-bit type `int`. However, integers whose absolute value is smaller than 2^{24} can be represented in an exact way in the type `float`.

A number x could potentially be represented by different (s, m, e) triples. However, it has many computational advantages to restrict the floating-point numbers to a normal form. Therefore, m is always chosen such that $1 \leq m < 2$ wherever possible (these are called the *normal numbers*) and

$0 \leq m < 1$ only were necessary (these are called the *subnormal numbers*). Note that, for subnormal numbers, $e = e_{min}$. Also note that, for subnormal numbers, the relative rounding error can be much worse than for normal numbers. On the other hand, the existence of subnormals guarantees the following equivalence: $(x > y) \leftrightarrow (x - y > 0)$. Without subnormals, this would not hold, because the difference between two normal floating-point numbers can be smaller than any normal floating-point number.

The triple representation gives us two zeros, $+0$ and -0, represented by $(0, 0, 0)$ and $(1, 0, 0)$, respectively.[2] The Boolean expressions $+0.0$==-0.0 and -0.0==$+0.0$ both return `true`. If the absolute value of the ideal result of a computation is too small to be representable as a floating-point number of the given format, the resulting floating point number is $+0$ or -0. In addition, there are three special values, $+\infty$, $-\infty$, and NaN (Not a Number). If the absolute value of the ideal result of a computation is too big to be representable as a floating-point number of the given format, the result is $+\infty$ or $-\infty$. Also, division by zero will give an infinite result (e.g., $7.3/+0 = +\infty$).[3] Computing further with infinity may give an infinite result (e.g., $+\infty + +\infty = +\infty$), but may also result in the additional 'error value' NaN (e.g., $+\infty - +\infty =$ NaN). The predicates `==`, `<`, `>`, `<=`, `>=` return `false` as soon as one operand evaluates to NaN. The predicate `!=` returns `true` as soon as one operand evaluates to NaN. In particular, NaN==NaN returns `false`, and NaN`!=`NaN returns `true`. Due to the presence of infinities and NaN, floating-point operations do *not* throw Java exceptions.[4]

By default, the Java virtual machine is allowed to make use of higher-precision formats provided by the hardware. This can make computation more accurate, but it also leads to platform-dependent behavior. This can be avoided by using the `strictfp` modifier, ensuring that floating-point computations inside methods or classes with the `strictfp` modifier comply to the precision defined in the IEEE 754 Standard, even for intermediate results. This modifier ensures portability. Additionally, one can set the *Assume strictfp* option in KeY, meaning that all computations are assumed to be within the scope of a `strictfp` modifier, even if `strictfp` is not explicit in the code.

[2] More precisely, the second element of these tuples has one zero before the radix point, and $p - 1$ zeros after the radix point.

[3] Note that $7.3/+0$ and $7.3/-0$ give different results, $+\infty$ and $-\infty$, respectively, even if $+0.0$==-0.0 returns `true`.

[4] The notion of exceptions in the IEEE 754 Standard is not to be confused with Java exceptions. IEEE 754 exceptions are rather 'sticky flags' for overflow, underflow, and other problematic situations. They do not change the control flow and certainly are not objects. In this context, we can also mention that IEEE 754 exceptions are not supported by Java.

3 Floating-point support in KeY

3.1 Arithmetics

In order to be able to specify and verify programs containing floating-point numbers, we made several extensions to the KeY tool. We added the `float` and `double` types to the KeY type system, introduced functions and predicate symbols to formalize arithmetic operations (`+`, `*`, ...), and comparisons (`<`, `==`, ...) on floating-point expressions, and added cast operations among floating-point and integer types (`(double)`, `(float)`, `(int)`). The translation supports both code with and without the `strictfp` modifier. However, since the actual precision of non-strictfp operations is not known, the function symbols remain uninterpreted. We assume that the `strictfp` modifier is set for all our benchmarks. We extended KeY's parser to correctly handle programs and annotations containing floating-point numbers and added logic rules for translating floating-point expressions from Java or JML to JavaDL.

As an example, Listing 1 shows a JML specification of our `Rectangle` benchmark that contains floating-point literals and makes use of the `fp_nan` and `fp_nice` predicates. `fp_nan` states that a floating-point expression is NaN and `fp_nice`, states that a floating-point expression is neither NaN nor infinity. The `scale` method contains two contracts that are checked separately, ensuring that the class fields of a scaled rectangle object are not NaN, considering different preconditions. For the first contract, the SMT solver produces a counterexample. In the second, we bound inputs by concrete ranges that we picked arbitrarily and get the valid result. In practice, such ranges would come from the context, e.g., from the kind of rectangles that appear in an application, or from known ranges of sensor values.

Concerning discharging the resulting proof obligations, there were two main ways to consider. One is to create a floating-point theory within KeY by adding axioms and deduction rules, so that the desired properties can be proven in KeY's calculus. The other way is to translate the proof obligations from JavaDL to SMT-LIB and call an external SMT solver. While the KeY approach traditionally favors conducting proofs with KeY's own, rule-based reasoning engine, we partially deviated from this way in order to harness the greater efficiency of SMT solvers when it comes to the combinatorically heavy reasoning about floating-point arithmetic. Our approach attempts to get the best of both worlds by distinguishing between basic floating-point arithmetic, i. e., elementary operations and comparisons, and more complex functions which do not have an SMT-LIB equivalent (e. g., the transcendental functions), or where rule-based reasoning is more effective than SMT solvers currently are (see Sect. 3.2.2).

Listing 1 The Rectangle.scale benchmark

```
/*@ public normal_behavior
  @  requires \fp_nice(arg0.x) && \fp_nice(arg0.y)
  @    && \fp_nice(arg1) && \fp_nice(arg2);
  @  ensures !\fp_nan(\result.x) && !\fp_nan(\result.y) &&
  @   !\fp_nan(\result.width) && !\fp_nan(\result.height);
  @ also
  @ public normal_behavior
  @  requires -5.53 <= arg0.x && arg0.x <= -3.38 &&
  @    -5.53 <= arg0.y && arg0.y <= -3.38 &&
  @    3.1 < arg0.width && arg0.width <= 3.7332 &&
  @    3.0000001 < arg0.height && arg0.height <=4.0004 &&
  @    3.0003001 < arg1 && arg1 <= 4.0024 &&
  @    -6.4000003 < arg2 && arg2 <= 3.0001;
  @  ensures !\fp_nan(\result.x) && !\fp_nan(\result.y)&&
  @   !\fp_nan(\result.width) &&!\fp_nan(\result.height);
  @*/
public Rectangle scale(Rectangle arg0,double arg1,double arg2){
  Area v1 = new Area(arg0);
  AffineTransform v2 =
    AffineTransform.getScaleInstance(arg1, arg2);
  Area v3 = v1.createTransformedArea(v2);
  Rectangle v4 = v3.getRectangle2D();
  return v4;
}
```

Elementary operations and comparisons get translated to the corresponding SMT-LIB functions. In SMT-LIB, all floating-point computations conform to the IEEE 754 Standard. Therefore, only Java programs with the `strictfp` modifier can be directly translated to SMT-LIB without loss of correctness.

We developed a translation from KeY's floating-point theory to SMT-LIB. In order to integrate it into KeY, we also overhauled the existing translation from JavaDL to SMT-LIB to create a new, more modular framework, which now supports all the features of the original translation, e. g., heaps and integer arithmetic, but also floating-point expressions at the same time.

Floating-point intricacies sometimes require extra caution. For example, there are two different notions of equality for floats: bitwise equality and IEEE754 equality. Our implementation ensures these are distinguished correctly and that the specification language remains intuitive for a developer to use.

Using the translation to SMT-LIB, we can specify and prove two classes of properties in KeY, the absence of special values and functional properties. The absence of special values is specified using the `fp_nan` and `fp_infinite` predicates (or the `fp_nice` equivalent). Furthermore, one can specify *functional* properties (including loop invariants) that are expressible in floating-point arithmetic, e.g., one can compare the result of a computation against the result of a different program which is known to produce a good result or a reference value.

3.2 Transcendental functions

Floating-point decision procedures in SMT solvers successfully handle programs consisting of arithmetic and square-root operations. Many numerical real-world programs, however, include transcendental functions such as `sin` and `cos`. In Java programs, these functions are implemented as static library functions in the class `java.lang.Math`.

Unlike arithmetic operations, transcendental functions are much more loosely specified by the IEEE 754 Standard—only an upper bound on the roundoff error is given. Libraries are thus free to provide different implementations, and even tighter error bounds. Exact reasoning in the same spirit as floating-point arithmetic would thus have to encode a specific implementation. Given that these implementations are highly optimized, this approach would be arguably complex. We observe, however, that such exact reasoning about transcendental functions is often not necessary and a sound approximate approach is sufficient and efficient.

In this section, we introduce an axiomatic approach for reasoning about programs containing transcendental functions. We observe that with the flexibility of deductive verification and KeY itself, we can instantiate it in two different ways. We encode transcendental functions as uninterpreted functions and axiomatize them in the SMT queries. Alternatively, we encode these axioms in KeY as logical inference rules. In the following, we explain each of these solutions in more detail and later we will evaluate them on a set of benchmarks.

3.2.1 Axiomatization in SMT

We encode library functions as uninterpreted functions and include a set of axioms in the SMT-LIB translation for each method that is called in a benchmark. That is, we extended KeY such that when a transcendental function appears in the proof obligation, its declaration alongside all the axioms for that function is added to the translation.

For the axiomatization of transcendentals, we did *not* add rules that expand to a definition or allow a repeated approximation of the function value (like expansion into a Taylor series). Instead, we added a number of lemmata encoding interesting properties related to special values. For instance, the following axiom states that if the input to the `sin` function is not a NaN or infinity, then the returned value of `sin` is between -1.0 and 1.0:

```
(assert (forall ((a Float64)) (=>
  (and (not (fp.isNaN a)) (not (fp.isInfinite a)))
  (and (fp.leq (sinDouble a)
               (fp #b0 #b01111111111#b0000...000000))
       (fp.geq (sinDouble a)
               (fp #b1 #b01111111111#b0000...000000))))))
```

Note that this implies that the result is not a NaN or infinity. The other axioms are similar in spirit, so we do not list them.

These axioms are expressed as quantified floating-point formulas and capture high-level properties of library functions complying with the specifications in the IEEE 754 Standard. Clearly, since we do not have the actual implementations of these functions, we are not able to prove arbitrary properties. However, such an axiomatization is often sufficient to check for (the absence of) special values, i.e., NaN and infinity, as our experiments in Sect. 4.4 show.

3.2.2 Taclets in KeY

Reasoning about quantified formulas in SMT is a long-lasting challenge [32]. We have also observed in our experiments with only arithmetic operations (Sect. 4.3) that SMT solvers struggle with quantifiers in combination with floating-point numbers. We have therefore implemented an alternative approach encoding the axioms not in the SMT queries, but instead as deductive inference rules (called 'taclets') in KeY.

The rules encode the same logical information as the universally quantified assertions that we add in SMT-LIB (and where we leave the choice of instantiations entirely to the SMT/SAT solver). With our taclet approach, we instantiate a quantifier (only) to one's needs. For instance, the taclet that corresponds to the SMT axiom mentioned in Sect. 3.2.1 is captured by the following taclet:

$$\begin{array}{ll} \texttt{find} & \sin(x) \\ \texttt{add} & \neg\texttt{fp_nan(x)} \wedge \neg\texttt{fp_infinite(x)} \\ & \rightarrow -1.0 \leq \sin(x) \leq +1.0 \quad \Longrightarrow \end{array}$$

We note that this is an incomplete treatment of the formalized operations. Considering only some and not all possible quantifier instantiations buys us more closed proofs and shorter running times in some cases. However, it may also lead to spurious counterexamples (false positives) reported by the SMT solver in other cases.

A heuristic strategy applies the rules automatically using the occurrences of transcendentals as instantiation triggers. However, instantiating the axioms too eagerly considerably increases the number of open goals, which is why we assume that the user selects the axioms to apply manually (and did so in the experiments). After the application, the proof obligation can either be closed, i.e., proven, by KeY automatically, or be given to the SMT solver as before for final solving. Currently, the set of axioms (in the SMT-LIB translation and as taclets in KeY) only contains axioms for the transcendental functions occurring in our benchmarks, namely sin, cos, and atan functions. So far we have 10 axioms; however, adding more axioms (also for further transcendentals like exponentiation or logarithm) is straightforward.

- If `arg` is NaN or an infinity, then `sin(arg)` is NaN.
- If `arg` is zero, then the result of `sin(arg)` is a zero with the same sign as `arg`.
- If `arg` is not NaN or infinity, then the returned value of `sin` is between -1.0 and 1.0.
- If `arg` is not NaN or infinity, then the returned value of `sin` is not NaN.
- If `arg` is NaN or an infinity, then `cos(arg)` is NaN.
- If `arg` is not NaN or infinity, then the returned value of `cos` is between -1.0 and 1.0.
- If `arg` is not NaN or infinity, then the returned value of `cos` is not NaN.
- If `arg` is NaN or an infinity, then `atan(arg)` is NaN.
- If `arg` is zero, then the result of `atan(arg)` is a zero with the same sign as `arg`.
- If `arg` is not NaN, then the returned value of `atan` is between $-\pi/2$ and $\pi/2$.

Fig. 1 List of added axioms to KeY for transcendental functions

The full set of axioms is listed in Fig. 1 (stated in natural language for presentation purposes).

3.3 Interaction of floats with other primitive types

For the formal verification of Java programs with floats, there are often not only floating-point numbers that have to be taken into consideration, but also heap data structures, arrays, and primitive values of other data types than `float` or `double`. Our SMT translation in KeY does not only target the floating-point theory, but encodes other aspects of proof obligations into theories like linear integer arithmetic for integers, or arrays representing heap data structures. In the evaluation, the state-of-the-art SMT solvers show their ability to reason about these theories when they occur simultaneously within the same proof obligation. As long as the different theories act on disjoint domains, this combined integrated translation works nicely in most cases. For instance, programs using `float` arrays or class attributes of type `double` can be handled as long as floating-point values and other types are not arithmetically combined within an expression.

However, there are relevant cases in which arithmetic expressions in Java programs have to contain both integer and floating-point variables, their domains thus becoming entangled. It is, for instance, a legal Java expression to add an integer value to a `float` value, yielding a `float` value.

Such cases of intersecting value domains require special attention.

We call *float-representable* those 32-bit `int` values which can be represented in `float` without loss of precision. In particular, all integers between -2^{24} and 2^{24} are float-representable.[5] According to the Java Language Specification (JLS) [33], an implicit widening cast called *numeric promotion* is applied when different primitive types are combined in an arithmetic operation. The addition `iv + fv` of an integer variable `iv` and a `float` variable `fv` is hence equivalent to the expression `(float)iv + fv` in which the numeric promotion has been made explicit.

The cast operator connects the domains of the two primitive types and maps integer values to floats by applying the same rounding operation which is applied to the result of arithmetic operations within `floats`. Since this family of cast operators on primitive types is translated from JML into JavaDL as uninterpreted functions, we can build a bridge between the integer and floating-point domains by adding theorems/axioms about the numeric promotion to KeY in form of taclets.

In KeY, we can add knowledge about the casting operator in form of taclet rules to the verification engine: The new taclets are implemented as conditional rewriting taclet rules introducing new constraints on the cast operator '`(float)`'.

A central property of float-representable integers is that the cast operation can be inverted, captured by the following taclet

```
find    (int)(float)i
replace i
show    -2^24 ≤ i ≤ 2^24
```

in KeY, formalising that casting an integer expression i first to a `float` and then back to an `int` is the identity if i stays within the range of float-representable values.

Since the JLS prescribes the same rounding mode for float operations and numeric promotion, it does not matter, for float-representable values, if they are first added and then cast or the way around:

Theorem 1 *Let a, b be float-representable integer values such that $a + b$ does not `int`-overflow. Then,* $(\texttt{float})(a + b) = (\texttt{float})a + (\texttt{float})b$.

Proof e introduce an auxiliary injection function $R : \texttt{float} \cup \texttt{int} \to \mathbb{R}$ canonically mapping float and integer values to their real-valued counterparts. The function $rnd : \mathbb{R} \to \texttt{float}$ is the rounding function mapping real values to floats. According to the JLS, the same rounding function is used after floating-point addition ($f_1 +_{\texttt{float}} f_2 = rnd(R(f_1) +_{\mathbb{R}} R(f_2))$ for floats f_1, f_2) and when converting integers to floats ($(\texttt{float})i = rnd(R(i))$ for an integer i).

With a, b as required by the theorem, we have $R(rnd(R(a))) = R(a)$, $R(rnd(b)) = R(b)$ and $R(a +_{\texttt{int}} b) = R(a) +_{\mathbb{R}} R(b)$. Hence,

$$
\begin{aligned}
&(\texttt{float})(a +_{\texttt{int}} b)\\
&= rnd(R(a +_{\texttt{int}} b)) = rnd(R(a) +_{\mathbb{R}} R(b))\\
&= rnd(R(rnd(R(a)) +_{\mathbb{R}} R(rnd(R(b))))\\
&= (\texttt{float})a +_{\texttt{float}} (\texttt{float})b
\end{aligned}
$$

Note that Theorem 1 does *not* require $a + b$ to be float-representable. Indeed, $a + b$ may need to be rounded when casted to `float` on the left of the equation. At the same time, the `float` addition on the right side of the equation may also require rounding. The insight from the proof above is that the IEEE standard and the JLS force the rounding on both sides to be the same!

Theorem 1 can also be formulated for subtraction and multiplication (with essentially the same correctness arguments) instead of addition. A taclet that implements this theorem for the float-representable numbers in the aforementioned range is the following:

```
find    (float)(a + b)
replace (float)a + (float)b
show    -2^24 ≤ a ≤ 2^24 ∧ -2^24 ≤ b ≤ 2^24
```

Note that the taclet does not cover all float-representable integers but only the interval in which all integers are float-representable. In this range, the sum $a + b$ cannot overflow. The taclet can also be formulated for subtraction. The taclet for the case of multiplication must take this into consideration:

```
find    (float)(a * b)
replace (float)a * (float)b
show        -2^24 ≤ a ≤ 2^24 ∧ -2^24 ≤ b ≤ 2^24
          ∧ -2^31 ≤ a * b < 2^31
```

Note that KeY internally operates on mathematical integers (instead of 32-bit integers) such that the overflow check can be implemented in this fashion.

Another important rule for the cast operator is the conversion of literals: any expression $(\texttt{float})c$ where c is a numerical integer literal can be rewritten to the corresponding numerical literal on the `float` side, like

[5] This interval is indeed the largest in which all integers are `float`-representable. The value $2^{24} + 1$ requires a 25-bit significand to be precisely represented, but `float` only has a precision of 24 bits.

Listing 2 Adding up the values in an array of floats.

```
/*@ public normal_behaviour
  @  requires (\forall int i; 0 <= i <a.length;
  @                1.f <= a[i] && a[i]<= 2.f);
  @  requires a.length < 16_777_216; //== 2^24
  @  ensures (float)a.length <= \result &&
  @          \result <= (float)a.length* 2.f;
  @*/
float sum(float[] a) {
    float sum = 0.0f;
    int i = 0;
    /*@ loop_invariant 0 <= i && i <= a.length &&
      @   (float)i <= sum && sum <= (float)i * 2.f;
      @ decreases a.length - i;
      @ assignable \nothing;
      @*/
     while(i < a.length) {
        sum += a[i];
        i++;
     }
     return sum;
}
```

```
find    (float)0
replace 0.0f          .
```

Similar theorems and rules can be formulated for other combinations of primitive types, in particular involving `double` and `long`.

An example program, which makes use of numeric promotion in Java, is shown in Listing 2, where the loop invariant and the postcondition contain a numeric promotion (made explicit in the listing). In order to prove that the loop invariant is inductive, one has to show that $(\texttt{float})(i+1) = (\texttt{float})i + 1.f$, which can be proven using the newly introduced taclet mentioned above.

4 Evaluation

4.1 Benchmark programs

We collected a set of existing floating-point Java programs representing real-world applications in order to evaluate the feasibility and performance of KeY's floating-point support.

The left half of Table 1 provides an overview of our benchmarks. Each benchmark consists of one method, which is composed of arithmetic operations and method calls to potentially other classes. The invocations of methods from `java.lang.Math` (e.g., `Math.abs`) are marked by "+1" in Table 1; these are resolved by inlining the method implementation. For benchmarks that contain calls to transcendental functions and square root, the called functions are listed; these are handled by our axiomatization. We include `sqrt` in this list, as we have observed that exact support can be expensive, so it may be advantageous to handle `sqrt` axiomatically. We include benchmarks with loops and loop invariants used for the evaluation of the tool `Pine` [39], represented as `Pine.*`. Benchmarks `Rectangle`, `Circuit`, `Matrix3`, `Rotation` and `Pine.pendulum-approx` are partially shown in Listings 1, 6, 3, 4 and 5 respectively.

Each benchmark also includes a JML contract that is to be checked. For some methods, we specify two contracts (marked by "(2)" in the first column of Table 1), each serving as an independent benchmark. The contracts for some of these benchmarks check that the methods do not return a special value, i.e., infinity and/or NaN, the preconditions being that the variables are not themselves special values and possibly are bounded in a given range. For the `Matrix`, `FPLoop`, `Rotate` and `Pine.*` benchmarks, we check a *functional* property (see Sect. 4.3). The `Pine.*` benchmarks and `FPLoop`, which has three contracts, additionally show how to specify floating-point loop behavior using loop invariants.

4.2 Proof obligation generation

To reason about the contract of a selected benchmark, we apply KeY, which generates proof obligations or 'goals'. Some of these goals (heap-related) are closed by KeY automatically. The remaining open goals are closed by either SMT solvers with floating-point support directly (Sects. 3.1 and 3.2.1), or with a combination of transcendental KeY taclets and floating-point SMT solving (Sect. 3.2.2).

Columns 6 and 7 in Table 1 show the number of proof obligations closed by KeY directly and to be discharged by external solvers, respectively. The next two columns show the number of taclet rules that KeY applied in order to close its goals, and the time this takes. For benchmarks with two contracts, we show the respective values separated by '/'.

We run our experiments on a server with 1.5 TB memory and 4x12 CPU cores at 3 GHz. However, KeY runs single-threadedly and does not use more than 8GB of memory.

For our set of benchmarks, the symbolic execution process is fully automated. Note that the machinery can deal with loop invariants, if they are provided. Automated loop invariant generation is, however, particularly challenging for floating-points due to roundoff errors [25,39], and a research topic in itself.

4.3 Evaluation of SMT floating-point support

Previous work [30] reported that SMT support for floating-point arithmetic is rather limited. However, with recent advances [15], we evaluate the situation again. Most benchmarks used to evaluate SMT solvers' decision procedures [55] aim to check (individual) specialized (corner case) properties of floating-point arithmetic. The proof obligations

Table 1 Benchmark details and KeY automode statistics, time is measured in seconds

Benchmark	Benchmark details				Automode statistics			
	# Classes	# Method calls	# Arith. ops	Library functions	# Goals closed by KeY	# Goals to be closed externally	# Rules applied	Automode time (s)
Complex.add (2)	1	0	2	–	3 / 3	1 / 4	185 / 286	0.7 / 0.2
Complex.divide (2)	1	0	11	–	10 / 8	2 / 8	483 / 625	0.7 / 0.8
Complex.compare	1	0	2	–	3	2	216	0.2
Complex.reciprocal (2)	1	1	6	–	1 / 1	2 / 2	402 / 406	0.4 / 0.5
Circuit.impedance	2	1	3	–	1	4	360	0.5
Circuit.current (2)	2	3	14	–	11 / 11	4 / 1	1267 / 1238	4.0 / 4.1
Matrix2.transposedEq	1	3	3	–	3	1	735	0.9
Matrix3.transposedEq	1	4	34	–	3	1	1786	5.1
Matrix3.transposedEqV2	1	4	34	–	3	1	1796	5.4
Rectangle.scale (2)	3 + 1	23	22	–	32 / 32	32 / 16	5990 / 5617	18.4 / 14.5
Rotate.computeError	1 + 1	6	26	–	108	8	3693	74.2
Rotate.computeRelErr	1 + 1	6	28	–	120	8	3898	79.6
FPLoop.fploop	1	0	1	–	2	4	99	0.1
FPLoop.fploop2	1	0	1	–	2	4	99	0.1
FPLoop.fploop3	1	0	1	–	2	4	99	0.1
Pine.ex2	1	1	9	–	2	12	320	1.3
Pine.ex2-reset	1	1	9	–	2	16	394	1.8
Pine.ex3-reset-leadlag	1	1	6	–	2	15	240	0.5
Pine.ex7-dampened	1	1	5	–	2	10	203	0.6
Pine.filter-goubault	1	1	3	–	2	8	183	0.5
Pine.filter-mine1	1	1	4	–	2	10	211	0.6
Pine.filter-mine2	1	1	3	–	2	8	172	0.6
Pine.filter-mine2-nondet	1	1	4	–	2	8	193	0.5
Pine.harmonic	1	1	5	–	2	9	199	0.4
Pine.pendulum-approx	1	1	17	–	2	10	299	0.9
Pine.pendulum-small	1	1	7	–	2	10	224	0.7
Pine.symplectic	1	1	6	–	2	9	195	0.7
Cartesian.toPolar	2 + 1	3	6	sqrt, atan	1	4	438	0.5
Cartesian.distanceTo	1 + 1	1	5	sqrt	2	1	191	0.1
Polar.toCartesian	2 + 1	3	4	cos, sin	1	2	364	0.5
Circuit.instantCurrent	2 + 1	14	23	sqrt, atan, cos	17	2	1686	14.1
Circuit.instantVoltage	1 + 1	1	4	cos	0	2	138	0.1

generated from our set of benchmarks are complementary in that they are more arithmetic heavy, while nonetheless relying on accurate reasoning about special values and functional properties.

For each open goal not automatically closed, KeY generates one SMT-LIB file that is fed to the solvers for validation. We compare the performance of the three major SMT solvers with floating-point support CVC4 [6] (version 1.8, with the SymFPU library [15] enabled), Z3 (4.8.9) [26] and MathSAT (5.6.3) [20]. For this, we set a timeout of 300s for each proof obligation. While KeY is able to discharge proof obligations in parallel, for our experiments, we do so sequentially to maintain comparability.

KeY's default translation to SMT introduces axioms which include quantifiers. These quantifications are not related to floating-point arithmetic, but are used to logically encode important properties of the Java memory model, like the type hierarchy and the absence of dangling references on any valid Java heap. If we reason about floating-point problems in isolation, they are not needed, but if we want to consider Java verification more holistically with questions combining aspects of heap and floating point reasoning,

Table 2 Summary of valid / invalid goals correctly decided and average running times of each solver for the SMT translations with and without quantified axioms

Details	Experiment	Quantified axioms	# Goals	CVC4		Z3		MathSAT	
				# Goals decided	Avg.	# Goals decided	Avg.	# Goals decided	Avg.
Table 3	Valid	✓	80	79	5.4	25	25	–	–
Table 4	contracts	✗	80	79	5.3	52	63.7	80	10.1
Table 5	Invalid	✓	9	0	7.6	0	2.6	–	–
	contracts	✗	9	8	12.8	7	60.4	9	2.0
Table 8	Axioms in SMT	✓	10	9	17.3	4	65.0	–	–
	Axioms as taclets	✓	10	10	28.0	2	164.0	–	–
	Axioms as taclets	✗	10	10	28.1	5	84.3	8	1.5
Table 9	fp.sqrt	✗	7	7	40.4	1	23.5	5	1.2
	Axiomatized sqrt	✗	7	5	2.1	5	86.4	5	3.9
Table 6	Loop invariants	✗	113	113	6.5	112	13.5	113	5.0

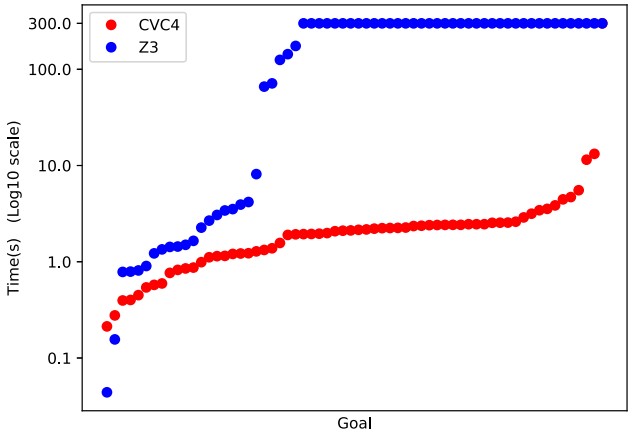

Fig. 2 Runtimes for valid goals with SMT translations *with* quantifiers

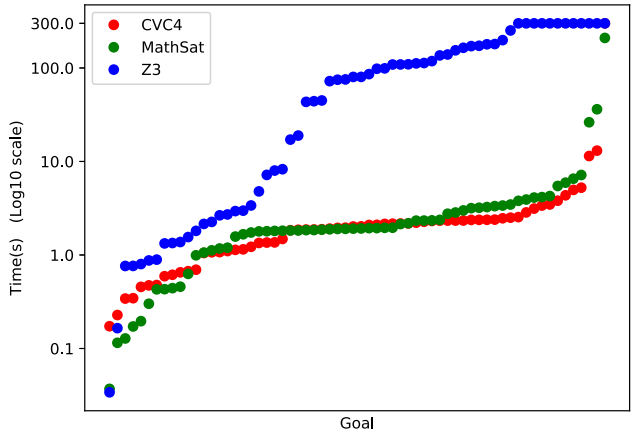

Fig. 3 Runtimes for valid goals with SMT translations *without* quantifiers

they become essential. We manually inspected that the proof obligations without our axiomatized treatment of transcendental functions do not depend on these properties and investigate the quantifier support by including or removing them from the SMT translations. We do not report results with quantifiers for MathSAT, since it does not support them.

Table 2 summarizes the results of our experiments; the first column lists the corresponding table with detailed results. Column 4 shows the number of expected valid or invalid goals for all benchmarks. For each solver, we show the number of goals that each solver can validate or invalidate, together with the average time (in seconds) needed. The goals resulting in timeout were excluded from the computation of the average time. Column 3 shows whether the SMT queries include quantifiers or not. The rows summarize our experiments with valid contracts, invalid contracts, axiomatizations of transcendental functions and square root, and loop invariants, respectively.

Figures 2 and 3 show a more detailed view of the solvers' running time for the valid benchmarks. The x-axis shows the number of open goals that are discharged by the SMT solvers, sorted by running time for each solver individually. The k-th point of one graph shows the minimum running time needed by the solver to close each of the k fastest goals. Note that each solver may have different goals which are its k fastest. The y-axis shows the time on a logarithmic scale, and the maximum value of 300 indicates the timeout.

We conclude that in the presence of quantified axioms and floating-point arithmetic solvers' performance deteriorate for both valid and invalid goals. In particular, none of the solvers is able to find counterexamples for any of the invalid goals. However, when the quantified axioms are removed

Listing 3 The Matrix3 benchmark

```
public class Matrix3 {
  //The matrix: [[a b c],[d e f],[g h i]]
  double a, b, c, d, e, f, g, h, i;
  double det;
  // method transpose not shown

  double determinant() {
    return (a * e * i + b * f * g + c * d * h) -
    (c * e * g + b * d * i + a * f * h);
  }

  double determinantNew() {
    return (a * (e * i) + (g * (b * f) + c * (d * h))) -
    (e * (c * g) + (i * (b * d) + a * (f * h)));
  }

  /*@ ensures \fp_normal(\result) ==> (\result == det); @*/
  double transposedEq() {
    det = determinant();
    return transpose().determinant();
  }

  /*@ ensures \fp_normal(\result) ==> (\result == det); @*/
  double transposedEqV2() {
    det = determinantNew();
    return transpose().determinantNew();
  }
}
```

from the SMT translations, their performance improves. For valid contracts, CVC4 and MathSAT perform better than Z3, in terms of both number of goals validated and the running time per goal. In particular, MathSAT is able to prove all goals. However, the running time performance of CVC4 is better than MathSAT's. For invalid contracts, solvers are able to produce the expected counterexamples at least partially. Particularly, MathSAT has a better performance than CVC4 and Z3 in terms of both running time and the number of proof obligations for which it can produce counterexamples.

Proving functional properties Listings 3 and 4 show examples of functional properties that are concerned with floating-point computation. The verification results are included in Table 3 and Table 4.

Table 3 Summary of valid goals proved and running times of each solver for the SMT translations *with* quantified axioms

Benchmark	# Goals	CVC4			Z3		
		# Goals proven	Avg.	Max.	# Goals proven	Avg.	Max.
Complex.add(1)	1	1	0.6	0.6	1	1.6	1.6
Complex.divide(1)	2	2	1.7	1.9	2	1.8	2.3
Complex.divide(2)	8	8	3.4	11.5	4	3.2	TO
Complex.compare	2	2	0.5	0.6	2	1.5	1.5
Complex.reciprocal(1)	2	2	1.0	1.7	0	–	TO
Complex.reciprocal(2)	2	2	1.8	2.4	2	2.7	4.0
Circuit.impedance	4	4	0.8	0.9	3	87.5	TO
Circuit.current(1)	4	4	6.3	13.2	0	–	TO
Circuit.current(2)	1	1	5.5	5.5	0	–	TO
Matrix2.transposedEq	1	1	0.5	0.5	0	–	TO
Matrix3.transposedEqV2	1	1	1.3	1.3	0	–	TO
Rectangle.scale(1)	32	31	2.2	TO	7	46.6	TO
Rotate.computeError	8	8	9.8	13.9	0	–	TO
Rotate.computeRelErr	8	8	25.3	45.6	0	–	TO
FPLoop.fploop	4	4	0.5	1.0	4	2.3	8.1
Summary	80	79	5.4	TO	25	25	TO

Table 4 Summary of valid goals proved and running times of each solver for the SMT translations *without* quantified axioms

Benchmark	# Goals	CVC4			Z3			MathSAT		
		# Goals proven	Avg.	Max.	# Goals proven	Avg.	Max.	# Goals proven	Avg.	Max.
Complex.add(1)	1	1	0.5	0.5	1	1.3	1.3	1	0.2	0.2
Complex.divide(1)	2	2	1.6	1.9	2	2.1	2.7	2	1.5	1.9
Complex.divide(2)	8	8	3.4	11.4	4	2.5	TO	8	29.0	209.5
Complex.compare	2	2	0.5	0.6	2	1.1	1.3	2	0.2	0.2
Complex.reciprocal(1)	2	2	0.9	1.5	1	198.5	TO	2	2.2	2.4
Complex.reciprocal(2)	2	2	1.7	2.3	2	2.8	3.4	2	1.5	1.9
Circuit.impedance	4	4	0.7	0.7	4	9.5	17.1	4	0.4	0.5
Circuit.current(1)	4	4	6.3	13.0	0	–	TO	4	13.3	36.2
Circuit.current(2)	1	1	5.2	5.2	0	–	TO	1	26.3	26.3
Matrix2.transposedEq	1	1	0.5	0.5	0	–	TO	1	0.6	0.6
Matrix3.transposedEqV2	1	1	1.1	1.1	0	–	TO	1	3.9	3.9
Rectangle.scale(1)	32	31	2.1	TO	32	95.1	251.8	32	2.4	4.2
Rotate.computeError	8	8	9.7	13.5	0	–	TO	8	22.7	35.7
Rotate.computeRelErr	8	8	25.0	45.0	0	–	TO	8	27.5	46.4
FPLoop.fploop	4	4	0.5	1.1	4	2.0	7.2	4	0.4	1.0
Summary	80	79	5.3	TO	52	63.7	TO	80	10.1	209.5

Table 5 Summary of invalid goals proved and running times of each solver for the SMT translations with and without quantified axioms

Benchmark	# Goals		CVC4				Z3				MathSAT			
			# Goals		Avg.	Max.	# Goals		Avg.	Max.	# Goals		Avg.	Max.
	Valid	Invalid	Valid	Invalid			Valid	Invalid			Valid	Invalid		
With quantified axioms														
Matrix3.transposedEq	0	1	0	0	–	TO	0	0	–	TO	–	–	–	–
Rectangle.scale(2)	12	4	12	0	12.2	TO	8	0	4.6	TO	–	–	–	–
Complex.add(2)	2	2	2	0	0.6	0.7	2	0	1.4	TO	–	–	–	–
FPLoop.fploop2	3	1	3	0	0.9	1.7	3	0	0.5	TO	–	–	–	–
FPLoop.fploop3	3	1	3	0	0.4	1.7	3	0	0.3	TO	–	–	–	–
Summary	20	9	20	0	7.6	TO	16	0	2.6	TO	–	–	–	–
Without quantified axioms														
Matrix3.transposedEq	0	1	0	1	170.2	170.2	0	0	–	TO	0	1	16.2	16.2
Rectangle.scale(2)	12	4	12	3	12.2	TO	12	3	108.2	TO	12	4	2.4	9.5
Complex.add(2)	2	2	2	2	0.5	0.5	2	2	0.7	1.0	2	2	0.2	0.2
FPLoop.fploop2	3	1	3	1	0.4	0.6	3	1	0.9	1.7	3	1	0.3	0.5
FPLoop.fploop3	3	1	3	1	0.3	0.6	3	1	0.6	1.7	3	1	0.2	0.4
Summary	20	9	20	8	12.8	TO	20	7	60.4	TO	20	9	2.0	16.2

For `Matrix`, we check that the determinants of a matrix and its transpose are equal. Note that this property holds trivially under real arithmetic, but not necessarily under floating-points. After feeding `transposedEq` (which uses the `determinant` method) and its contract to KeY, increasing the default timeout sufficiently and discharging the created goal, CVC4 generates a counterexample in 170.2s seconds and MathSAT in 16.2s. Z3 times out after 30 minutes. By feeding `transposedEqV2` (which uses the `determinantNew` method) to KeY, CVC4 validates the contract in 1.1s, MathSAT in 3.9s and Z3 times out again. One thing worth noting is that the way programs are written can greatly influence the computational complexity needed to reject or verify the contract. This is evident from the fact that slightly modifying the order of operations (using `determinantNew` instead) substantially reduces verification time and changes the verification result for MathSAT and CVC4.

Listing 4 The Rotation benchmark

```
public class Rotation {
  final static double cos90 = 6.123233995736766E-17;
  final static double sin90 = 1.0;

  // rotates a 2D vector by 90 degrees
  public static double[] rotate(double[] vec) {
    double x = vec[0] * cos90 - vec[1] * sin90;
    double y = vec[0] * sin90 + vec[1] * cos90;
    return new double[]{x, y};
  }

  /*@ requires (\forall int i; 0 <= i && i < vec.length;
    @  \fp_nice(vec[i]) && vec[i] > 1.0 && vec[i] < 2.0) &&
    @   vec.length == 2;
    @ ensures  \result[0] < 1.0E-15 &&
    @ \result[1] < 1.0E-15;
    @*/
  public static double[] computeError(double[] vec) {
    double[] temp = rotate(rotate(rotate(rotate(vec))));
    return new double[]{Math.abs(temp[0] - vec[0]),
     Math.abs(temp[1] - vec[1])};
  }
}
```

For `Rotate`, we check that the difference between an original vector and the one that is rotated four times by 90 degrees must not be larger than 1.0E-15. We also verified the same bound for the relative difference (by exploiting another method and contract) for this benchmark. The constant `cos90` in Listing 4 is not precisely 0.0 to account for rounding effects in the computation of the cosine. `FPLoop` includes three loops, for which the contracts check that the return value is bigger than a given constant.

Furthermore, we have investigated KeY's capability in proving floating-point loop invariants in more details in the next experiment.

Though not always very fast, these examples show that verification of functional floating-point properties is viable.

Proving loop invariants We conducted an experiment to assess KeY in proving floating-point loop invariants for a set of benchmarks. As part of this experiment, we verified some of the floating-point loop invariants generated by the Pine tool [39] (represented as `Pine.*` in Table 1). Listing 5 shows the nonlinear `Pine.pendulum-approx` benchmark and the loop invariant that Pine has generated. This benchmark simulates a simple pendulum and uses a Taylor approximation of the sine function. The condition of the while loop is a placeholder for any condition, meaning that we are proving the loop invariant regardless of how many iterations the loop takes. We thus specify **diverges true**, which means that the method is (unconditionally) allowed to not terminate.

Row 10 in Table 2 summarizes the results of this experiment, and Table 6 shows the detailed results. As shown, KeY is able to prove that all but one of the considered invariants are in fact inductive. We used the timeout of two hours for this experiment and observed that MathSAT performs better than the other solvers in proving the invariants.

Note that not all invariants generated by Pine are necessarily verifiable by KeY (e.g., `Ex2` in Table 6), because the semantics of Pine and KeY differ subtly. In Pine, the generated invariant and bounds are considered to be real-valued, whereas in KeY they are evaluated under a floating-point semantics.

Sensitivity to contract variations We conducted an experiment on our `Rectangle.scale` benchmark to assess the solver's sensitivity to various changes, applied to the benchmark's contract or its implementation.

Specifically, we considered the following versions of the benchmark:

- $v0$: is the original version of the benchmark (Listing 1 using the second contract) and our baseline;
- $v1$: reduces the number of classes involved to two, while keeping the same functionality;
- $v2$: reduces the number of classes involved to one, while keeping the same functionality;
- $v3$: modifies $v2$ such that variable bounds in the precondition become more "complicated" in terms of longer fractional parts (e.g., the bounds for `arg2` become [3.0000001, -6.4000000003] instead of [3.0001, -6.4000 003]);
- $v4$: simplifies the mathematical expression of $v2$ (less arithmetic operations)
- $v5$: modifies $v3$ such that `arg2` has a tighter bound, i.e., the interval width is smaller
- $v6$: modifies $v2$ such that `arg2` has a larger bound, i.e., the interval width is larger
- $v7$: modifies $v2$ such that only `arg2` has a "complicated" bound
- $v8$: modifies $v0$ such that `arg2` has a tighter bound

Table 7 summarizes the results for this experiment. With the quantified formulas included in the SMT translation, both CVC4 and Z3 are able to prove more goals when the number of classes is reduced, and also when the number of arithmetic operations is reduced. Z3 further seems to be sensitive to whether variable bounds are "complicated" or not, whereas CVC4 is not. We obtain a somewhat surprising result when `arg2` has a tighter bound. While Z3's performance improves, CVC4 validates two goals less. On the other hand, increasing the bounds on `arg2` does not seem to make a difference.

It seems that `arg2` is the bottleneck for this benchmark; when only `arg2` has a "complicated" input interval, CVC4 proves less goals. Finally, constraining `arg2` in the original

Table 6 Summary of goals proved and running times of each solver for benchmarks with floating-point loop invariants without quantified axioms

Benchmark	# Goals	CVC4			Z3			MathSAT		
		# Goals proven	Avg.	Max.	# Goals proven	Avg.	Max.	# Goals proven	Avg.	Max.
Ex2	12	9(CE)	5.2	34.4	9(CE)	27.6	TO	9(CE)	3.4	22.7
Ex2-reset	16	16	0.8	4.2	16	19.3	162.8	16	0.9	2.9
Ex3-reset-leadlag	15	15	0.7	5.7	15	0.4	496.8	15	0.9	8.5
Ex7-dampened	10	10	1.5	10.2	10	1.9	3081.0	10	3.9	34.2
Filter-goubault	8	8	2.1	13.2	8	8.7	360.4	8	3.3	21.8
Filter-mine1	10	10	1.2	4.6	10	1.3	522.2	10	1.9	8.1
Filter-mine2	8	8	8.4	63.0	8	0.3	997.5	8	21.0	163.1
Filter-mine2-nondet	8	8	0.9	2.7	8	14.1	707.7	8	0.8	3.6
Harmonic	9	9	2.9	20.1	9	28.5	2643.0	9	2.7	20.2
Pendulum-approx	10	10	47.2	223.7	9	4.9	TO	10	16.8	421.2
Pendulum-small	10	10	1.9	14.5	10	7.5	2412.5	10	1.8	14.6
Symplectic	9	9	1.7	9.2	9	32.9	1989.1	9	3.4	21.5
Summary	113	113	6.5	223.7	112	13.5	TO	113	5.0	421.2

Table 7 SMT solvers summary statistics for various versions of the `Rectangle` benchmark with quantified axioms in the SMT translations

version	Applied change	# Goals	CVC4			Z3		
			# Goals validated	Avg.	Max.	# Goals validated	Avg.	Max.
v0	None (original)	32	31	2.3	TO	7	46.2	TO
v1	Fewer classes (2) in v0	32	32	2.5	4.8	9	42.7	TO
v2	Fewer classes (1) in v0	32	32	2.4	4.2	7	85.2	TO
v3	Complicated intervals for all vars in v2	32	32	2.5	5.5	6	59.3	TO
v4	Simpler math in v2	16	16	1.0	1.3	12	35.3	TO
v5	Shorter interval for arg2 in v3	32	30	2.3	TO	9	95.1	TO
v6	Longer interval for arg2 in v2	32	32	2.6	7.4	7	14.8	TO
v7	Complicated interval for arg2 in v2	32	31	2.4	TO	7	23.9	TO
v8	Shorter interval for arg2 in v0	32	32	2.5	4.2	7	46.5	TO

benchmark more tightly allows CVC4 to validate all goals but Z3's performance remains unaffected.

With the SMT-LIB translation that does not introduce quantified axioms, we can see that CVC4's results, in terms of number of goals validated, are the same as before, while Z3 performs much better than before. MathSAT is able to validate all goals of all versions.

In summary, the solvers' performance seems to be sensitive to slight innocuous looking changes such as the number of classes involved and variable bounds. For example, constraining `arg2` in the original benchmark more tightly allows CVC4 to validate all goals (1 more). This behavior could be potentially exploited by, e.g., relaxing a variable's bounds.

4.4 Transcendental functions in KeY

We evaluated the two approaches from Sect. 3.2.1 on our set of benchmarks; rows 5, 6, and 7 in Table 2 summarize the results. The detailed results of these experiments are included in Table 8. Note that both approaches are fully automated.

We conclude that the SMT solvers perform better when the axiomatization is applied at the KeY level. When axioms for transcendental functions are added to the SMT-LIB translation directly Z3 validates 4 out of 10 goals. With the axiomatization at the KeY level, solvers are able to validate more goals (with quantified formulas removed from the SMT translations), e.g., Z3 is able to validate 5 goals and CVC4 can validate all. Therefore, it is preferable to apply them on the KeY side via taclet rules.

All the solvers we have used in this work comply with the IEEE 754 standard and therefore have bit-precise support for the square-root function. They provide bit-precise reasoning by effectively encoding the behavior of floating-point circuits over bitvectors (which is naturally expensive), together with different heuristics and abstractions to speed up solving time. However, depending on the property, we do not always need bit-precise reasoning, so we propose handling

Table 8 Summary statistics with axioms in SMT-LIB translations and as taclet rules in KeY

Benchmark	# Goals	CVC4			Z3			MathSAT		
		# Goals validated	Avg.	Max.	# Goals validated	Avg.	Max.	# Goals validated	Avg.	Max.
Axioms in SMT-LIB translation										
Cartesian.toPolar	4	4	7.1	9.2	1	16.8	TO	–	–	–
Polar.toCartesian	2	2	0.9	0.9	2	69.7	95.7	–	–	–
Circuit.instantCurrent	2	1	123.6	TO	0	–	TO	–	–	–
Circuit.instantVoltage	2	2	1.1	1.1	1	103.8	TO	–	–	–
Summary	10	9	17.3	TO	4	65.0	TO	–	–	–
Axioms as taclet rules in KeY with quantified formulas										
Cartesian.toPolar	4	4	6.8	7.7	1	40.0	TO	–	–	–
Polar.toCartesian	2	2	1.4	1.9	1	288.0	TO	–	–	–
Circuit.instantCurrent	2	2	123.8	128.3	0	–	TO	–	–	–
Circuit.instantVoltage	2	2	1.3	1.3	0	–	TO	–	–	–
Summary	10	10	28.0	128.3	2	164.0	TO	–	–	–
Axioms as taclet rules in KeY without quantified formulas										
Cartesian.toPolar	4	4	6.9	7.5	1	23.5	TO	4	1.2	1.7
Polar.toCartesian	2	2	1.5	2.3	2	52.6	81.2	2	0.6	0.8
Circuit.instantCurrent	2	2	123.5	127.5	0	–	TO	0	–	TO
Circuit.instantVoltage	2	2	1.5	1.7	2	146.4	160.7	2	0.8	0.8
Summary	10	10	28.1	127.5	5	84.3	TO	8	1.5	TO

Listing 5 The pendulum-approx benchmark

```
/*@ public normal_behavior
  @ requires 0.0f <= u && u <= 0.0f && 2.0f <= v && v <= 3.0f;
  @ diverges true;
  @*/
  public float pendulum-approx(float u, float v) {

    /*@ loop_invariant -1.1f <= u && u <= 1.2f  &&
      @  -3.2f <= v && v <= 3.1f &&
      @  (-0.11f*u) + (0.01f*v) + (1.0f*u*u) + (0.03f*u*v)
      @  + (0.12f*v*v) <= 1.15f;
      @*/
      while (true) {
       u = u + 0.01f * v;
       v = v + 0.01f * (-0.5f * v - 9.81f *
        (u - (u * u * u) / 6.0f +
        (u * u * u * u * u) / 120.0f));
      }
      return u;
}
```

the square-root function with the same taclet-based axiomatization as introduced in Sect. 3.2.2.

To this end, we conducted an experiment on the benchmarks containing `sqrt`, comparing the approach from Sect. 3.2.2 (adding the necessary axioms, resp. taclet rules) to using the square root implemented in SMT solvers (`fp.sqrt`). We chose to include only axioms specified in or inferred from the IEEE 754 standard. The set of used axioms are as follows:

- If `arg` is NaN or less than zero, then `sqrt(arg)` is NaN.
- If `arg` is positive infinity, then `sqrt(arg)` is positive infinity.
- If `arg` is positive zero or negative zero, then `sqrt(arg)` is the same as `arg`.
- If `arg` is not NaN and greater than or equal to zero, then `sqrt(arg)` is not NaN.
- If `arg` is not infinity and is greater than one then `sqrt(arg)` $<$ `arg`.

Rows 8 and 9 in Table 2 summarize the results for this experiment; the detailed results are included in Table 9.

We observed that for two out of the three benchmarks, the average running time of all solvers decreases using the axiomatized square root. Furthermore, Z3 is able to reason about more proof obligations with the axiomatized version. However, the success of this approach depends on the axioms added to KeY and may not always work if we do not have suitable axioms. For example, for the `Circuit.instantCurrent` benchmark which computes the instantanious current of an RL circuit (Listing 6), using the axiomatized square root, CVC4 is not able to validate the contract, but with `fp.sqrt` the contract is validated. The reason our approach is unsuccessful on this benchmark is that the square-root function appears early in the computation followed by atan and cos afterwards, resulting in complex

Table 9 Summary statistics for benchmarks containing the square-root function, with quantified formulas removed from the SMT-LIB translation

Benchmark	# Goals	CVC4			Z3			MathSAT		
		# Goals validated	Avg.	Max.	# Goals validated	Avg.	Max.	# Goals validated	Avg.	Max.
fp.sqrt										
Cartesian.toPolar	4	4	6.9	7.5	1	23.5	TO	4	1.2	1.7
Cartesian.distanceTo	1	1	8.2	8.2	0	–	TO	1	1.0	1.0
Circuit.instantCurrent	2	2	123.5	127.5	0	–	TO	0	–	TO
Summary	7	7	40.4	127.5	1	23.5	TO	5	1.2	TO
Axiomatized sqrt										
Cartesian.toPolar	4	4	2.0	2.9	4	49.81	163.0	4	1.0	1.6
Cartesian.distanceTo	1	1	2.7	2.7	1	233.0	233.0	1	1.0	1.0
Circuit.instantCurrent	2	0	–	TO	0	–	TO	0 (2 CE)	11.1	13.8
Summary	7	5	2.1	TO	5	86.4	TO	5	3.9	13.8

Listing 6 The Circuit.instantCurrent benchmark

```
public class Circuit {
  double maxVoltage, frequency, resistance, inductance;
  // ...

/*@ public normal_behavior
  @ requires 1.0 < this.maxVoltage && this.maxVoltage < 12.0 && 1.0 < this.frequency && this.frequency < 100.0 &&
  @  1.0 < this.resistance && this.resistance < 50.0 && 0.001 < this.inductance && this.inductance < 0.004 &&
  @  0.0 < time && time < 300.0;
  @ ensures !\fp_nan(\result) && !\fp_infinite(\result);
  @*/
public double instantCurrent(double time) {
  Complex curr = computeCurrent();
  double maxCurrent = Math.sqrt(curr.getRealPart() * curr.getRealPart() + curr.getImaginaryPart() * curr.getImaginaryPart());
  double theta = Math.atan(curr.getImaginaryPart() / curr.getRealPart());
  return maxCurrent * Math.cos((2.0 * Math.PI * frequency * time) + theta);
}
}
```

expressions. In order to prove the corresponding proof obligations, in this case stronger axioms are needed.

In summary, treating `sqrt` axiomatically can result in shorter solving times than performing bit-precise reasoning, but the approach may not always succeed when the axioms are not sufficient to prove a particular property.

4.5 Discussion and insights

In our set of experiments, we used benchmarks taken from real code to examine the feasibility of our approach and the solver's support for the floating-point theory. We can conclude that, for our set of benchmarks, generally all the solvers perform better with an SMT translation that does not introduce quantified formulas. Especially if the contract of the benchmark is invalid, the solvers are not able to produce counterexamples when quantifiers are present. Another observation is that solvers' performance is affected by the number of heap operations performed and the complexity of variable input ranges. From our experiment with programs containing loops, we observed that KeY is able to prove the provided invariants and solvers are mostly able to prove the generated goals. Finally, based on our results, we can confirm that the support for floating-point theory seams promising in all the solvers we examined. However, in terms of scalability and the size and type of problems they can handle, there is still room for improvement.

From our experiments with programs containing transcendental functions (and `sqrt`), we observed that handling these functions as uninterpreted with an appropriate axiomatization at the SMT-LIB or the taclet level is a viable approach, with an interesting trade-off between the verifier performance and the properties that can be proven. Clearly, properties that require exact semantics for transcendental functions will not be proven with our approach, but our experiments show that reasoning about the absence of special values is indeed possible. We further observe that both our approaches for adding axioms to the SMT queries or as

taclet rules work to some extent, however, applying axioms as taclets in KeY directly has several advantages. Using taclets avoids quantified axioms in the SMT query, which in turn improves the performance. While the additional assertions do not compromise the theoretical decidability of the theory (since the quantified domains are finite), they add considerably to the complexity of the encoding for the bit-blasting SAT-based decision procedures such that running times may increase exponentially. The experiments have thus exposed a weakness of the SAT-based verification approach: if symbols outside the canonical arithmetic operations with fixed semantics are used within a program or specification, providing semantics for these symbols within the SMT-LIB translation is not efficient.

Concretely, applying the axiomatization as taclets (and removing other quantifiers from the SMT translation) allows us to use MathSAT as the solver. Furthermore, we have also observed that quantifiers in the SMT queries result in poor performance (unknown results or timeouts) when a query is invalid. Applying axioms as taclets lets us avoid this issue. The rule-based sequent calculus, on which the KeY reasoning engine is based, deals with universally quantified symbol axiomatizations very successfully in many domains. The taclet mechanism and the implemented automatic strategy allow one to control the treatment of the symbol depending on a variety of side conditions, e.g., applying a lemma only if another relevant formula is also present in the proof obligation. Furthermore, taclets can also be applied manually, which allows the verifying user to control the calculus in a very fined-grained manner. Another advantage of having taclet rules is that we can create taclets with different formats for a single axiom and by doing so can sometimes even reduce the size of the proof obligation. This is the case when, instead of adding an axiom to the proof obligation, we replace a term with another one and thus avoid enlarging the proof obligation. That said, an axiomatization at the KeY level may result in spurious counterexamples if a rule application for an uninterpreted symbol, which would make the sequent valid, has not been applied yet. However, such a spurious counterexample can be straightforwardly identified by simply executing the method in question on it.

To summarize, the experiments show that highly automated floating point program verification is viable for relevant properties (handling of special values and some functional properties), up to a certain level of complexity (given by the SMT solvers). The choices of which parts of a proof obligation are delegated to SMT, and how they are translated to SMT are crucial for achieving effective and efficient program verification. Arithmetic operations proved to be more efficiently dealt with by delegation to SMT, whereas for transcendental functions, axiomatization and rule-based treatment in the theorem prover, outside the SMT solver, perform clearly better.

5 Related work

Our implementation uses the floating-point SMT-LIB theory [16], which, however, does not handle transcendental functions, as their semantics is (library) implementation dependent. Some real-valued automated solvers do handle transcendental functions [3,31], but to the best of our knowledge, the combination of floating-points and reals in SMT solvers is still severely limited.

None of the existing deductive verifiers support floating-point transcendental functions automatically. The Why3 deductive verification framework [28] has support for floating-point arithmetic, with front-ends for the C and Ada programming languages through Frama-C [23] and SPARK [17,30], respectively. Why3 has back-end support for different SMT solvers, as well as interactive proof assistants like Coq. Until recently, Why3 would discharge still many interesting floating-point problems with the help of Coq, relying on significant user interaction. In later work [30] (in the context with floating-point verification for Ada programs), Why3 can achieve a higher degree of automation. Note, however, that the user is still required to add code assertions as well as 'ghost code' to a significant extent.

The Boogie intermediate verification language [46] also supports floating-point expressions and targets Z3 for discharging proof obligations. In the Boogie community, it was observed that writing a specification in Boogie leads to decreases in SMT solver performance when compared to writing the goal in SMT-LIB directly, probably due to an inherent mixing of theories when using Boogie [57]. This matches our own experiences, and separation of theories should be considered an important task for the further development of floating-point verification.

Other deductive verifiers for Java have only rudimentary support for floating-points. Verifast [40] treats floating-point operations as if they were real values, and OpenJML [21] parses programs with floating-point operations, but essentially treats `float` and `double` as uninterpreted sorts.

The Java category of verification competition SV-COMP [10] contains a number of benchmarks that make use of floating-point variables. However, the focus of these benchmarks is usually not on arithmetical properties of expressions, but on the completeness of the Java language support. Amongst the participants of SV-COMP 2020, Symbolic (Java) Pathfinder (SPF) [54] (and various extensions) and the Java Bounded Model Checker (JBMC) [22] support floating-point arithmetic. Besides being limited to exploring the state space up to a bounded depth, their constraint languages do not support quantifiers and abstracting of method calls—which are features that we have used in this work.

Floating-point arithmetic has also been formalized in several interactive theorem provers [14,29,41]. While one can prove intricate properties about floating-point programs [12,

13,37], proofs using interactive provers are to a large part manual and require significant expertise.

Abstract interpretation-based techniques can show the absence of special values in floating-point code fully automatically, and several abstract domains which are sound with respect to floating-point arithmetic exist [18,42]. While the analysis itself is fully automated, applying it successfully to real-world programs in general requires adaptation to each program analyzed by end-users, e.g., the selection of suitable abstract domains or widening thresholds [11].

Besides showing the absence of special values, recent research has developed static analyses to bound floating-point roundoff errors [24,34,47,51,58]. These analyses currently work only for small arithmetic kernels and the tools in particular do not accept programs with objects.

Dynamic analyses generally scale well on real-world programs, but can only identify bugs (when given failure-triggering input), rather than proving correctness for *all* possible inputs. Executing a floating-point program together with a higher-precision one allows one to find inputs which cause large roundoff errors [9,19,43]. Ariadne [5] uses a combination of symbolic execution, real-valued SMT solving and testing to find inputs that trigger floating-point exceptions, including overflow and invalid operations. Our work subsumes this approach as the SMT solvers that we use can directly generate counterexamples, but more importantly, KeY is able to prove the absence of such exceptions.

6 Conclusion

In this work, we set out to enable efficient verification of programs which feature floating-point computations in combination with the other features of a fully fledged, widely used programming language, here Java. This is a different problem from verifying floating-point computations in isolation. To achieve that, we extend the verification tool KeY, which prior to this work supported full sequential Java except floating-point types. The core of KeY is a prover applying proof rules capturing an axiomatic semantics of the target language (Java), with the option to export proof goals to SMT solver plug-ins. This gave us the freedom to decide which features, and under which circumstances, should be dealt with by proof rules within the KeY prover, or by exporting goals to an SMT solver, respectively.

By joining the complementary strengths of SAT-based SMT solving and rule-based deduction, we presented the first working floating-point support in a deductive verification tool for Java. At the same time, we close a remaining gap in KeY to now support full sequential Java. Our evaluation shows that for specifications dealing with absence of NaN and infinity, as well as with value ranges, our approach can verify realistic programs automatically within a reasonable time frame. This includes programs using transcendental functions, as well as programs with loops. We observe that the MathSAT and CVC4 solver's floating-point support scales sufficiently for our benchmarks, as long as the queries do not include any quantifiers. On the other hand, our axiomatized approach for transcendental functions is best realized using calculus rules in KeY's internal reasoning engine, rather than in SMT solvers. We also presented rules for handling potentially rounding casts from integers to floating-point types.

While our work is implemented within the KeY verifier, we expect the insights from this work to be portable to other verifiers.

Acknowledgements This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) project 387674182. The authors would like to thank Daniel Eddeland, who together with co-author W. Ahrendt performed prestudies which impacted the current work.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive verification of floating-point java programs in key. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2021)
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, LNCS, vol. 10001. Springer (2016)
3. Akbarpour, B., Paulson, L.C.: MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. J. Autom. Reason. **44**(3) (2010)
4. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging rust types for modular specification and verification. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) (2019)
5. Barr, E.T., Vo, T., Le, V., Su, Z.: Automatic detection of floating-point exceptions. In: Principles of Programming Languages (POPL) (2013)
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Computer Aided Verification (CAV) (2011). Snowbird, Utah
7. Barrett, C., Stump, A., Tinelli, C., et al.: The SMT-LIB standard: version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (2010)

8. Beckert, B., Nestler, B., Kiefer, M., Selzer, M., Ulbrich, M.: Experience report: Formal methods in material science. CoRR arXiv:1802.02374 (2018)
9. Benz, F., Hildebrandt, A., Hack, S.: A dynamic program analysis to find floating-point accuracy problems. In: Programming Language Design and Implementation (PLDI) (2012)
10. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2020)
11. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Programming Language Design and Implementation (PLDI) (2003)
12. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: A comprehensive mechanized proof of a C program. J. Autom. Reason. **50**(4) (2013)
13. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for certifying floating-point programs. In: Intelligent Computer Mathematics (2009)
14. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: IEEE Symposium on Computer Arithmetic (ARITH) (2011)
15. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2019)
16. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for IEEE-754 floating-point arithmetic. In: IEEE Symposium on Computer Arithmetic (ARITH) (2015)
17. Chapman, R., Schanda, F.: Are we there yet? 20 years of industrial theorem proving with SPARK. In: Interactive Theorem Proving (ITP) (2014)
18. Chen, L., Miné, A., Cousot, P.: A sound floating-point polyhedra abstract domain. In: Asian Symposium on Programming Languages and Systems (APLAS) (2008)
19. Chiang, W.F., Gopalakrishnan, G., Rakamaric, Z., Solovyev, A.: Efficient search for inputs causing high floating-point errors. In: Principles and Practice of Parallel Programming (PPoPP) (2014)
20. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2013)
21. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: NASA Formal Methods (NFM) (2011)
22. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying java bytecode. In: Computer Aided Verification (CAV) (2018)
23. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Software Engineering and Formal Methods (SEFM) (2012)
24. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018)
25. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM Trans. Program. Lang. Syst. (TOPLAS) **39**(2) (2017)
26. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2008)
27. Eilers, M., Müller, P.: Nagini: A static verifier for python. In: Computer Aided Verification (CAV) (2018)
28. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European Symposium on Programming (ESOP) (2013)
29. Fox, A., Harrison, J., Akbarpour, B.: A formal model of IEEE floating point arithmetic. HOL4 Theorem Prover Library (2017). https://github.com/HOL-Theorem-Prover/HOL/tree/master/src/floating-point
30. Fumex, C., Marché, C., Moy, Y.: Automating the verification of floating-point programs. In: Verified Software: Theories, Tools, and Experiments (VSTTE) (2017)
31. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: International Conference on Automated Deduction (CADE-24) (2013)
32. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Computer Aided Verification (CAV) (2009)
33. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, Second Edition: The Java Series, 2nd edn. Addison-Wesley (2000)
34. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011)
35. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Asian Symposium on Programming Languages and Systems (APLAS) (2013)
36. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Handbook of Philosophical Logic, pp. 99–217. Springer (2001)
37. Harrison, J.: Floating point verification in HOL light: the exponential function. Formal Methods Syst. Des. **16**(3) (2000)
38. IEEE, C.S.: IEEE standard for floating-point arithmetic. IEEE Std 754-2008 (2008)
39. Izycheva, A., Darulova, E., Seidl, H.: Counterexample and simulation-guided floating-point loop invariant synthesis. In: Static Analysis Symposium (SAS) (2020)
40. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and java. In: NASA Formal Methods (NFM) (2011)
41. Jacobsen, C., Solovyev, A., Gopalakrishnan, G.: A parameterized floating-point formalizaton in HOL Light. Electron. Notes Theoret. Comput. Sci. **317** (2015)
42. Jeannet, B., Miné, A.: Apron: a library of numerical abstract domains for static analysis. In: Computer Aided Verification (CAV) (2009)
43. Lam, M.O., Hollingsworth, J.K., Stewart, G.W.: dynamic floating-point cancellation detection. Parallel Comput. **39**(3) (2013)
44. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes **31**(3) (2006)
45. Leavens, G.T., Cheon, Y.: Design by contract with JML (2006). http://www.jmlspecs.org/jmldbc.pdf
46. Leino, K.R.M.: This is Boogie 2 (2008). https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/
47. Magron, V., Constantinides, G., Donaldson, A.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. **43**(4) (2017)
48. Marché, C., Paulin-Mohring, C., Urbain, X.: The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. J. Logic Algebraic Programm. **58**(1) (2004)
49. McCormick, J.W., Chapin, P.C.: Building high integrity applications with SPARK. Cambridge University Press, Cambridge (2015)
50. Meyer, B.: Applying "Design by Contract". Computer **25**(10) (1992)
51. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.: Automatic estimation of verified floating-point round-off errors via static analysis. In: SAFECOMP (2017)
52. Muller, J., Brisebarre, N., de Dinechin, F., Jeannerod, C., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010)
53. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2016)

54. Pasareanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M.R., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: International Symposium on Software Testing and Analysis (ISSTA) (2008)
55. QF_FP SMT benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP (2019)
56. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Using model checking with symbolic execution to verify parallel numerical programs. In: International Symposium on Software Testing and Analysis (ISSTA) (2006)
57. Slow verification of programs combining multiple floating point values (Github issue) (2019 (accessed May 11, 2020)). https://github.com/boogie-org/boogie/issues/109
58. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In: Formal Methods (FM) (2015)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.