JConstraints: A Library for Working with Logic Expressions in Java

Falk Howar[1], Fadi Jabbour[2], and Malte Mues[2]

[1] Dortmund University of Technology and Fraunhofer ISST, Germany,
falk.howar@tu-dortmund.de
[2] Dortmund University of Technology, Germany

Abstract. In this paper we present JConstraints, a constraint solver abstraction layer for Java. JConstraints provides an object representation for logic expressions, unified access to different SMT and interpolation solvers, and useful tools and algorithms for working with logic formulas. The object representation enables implementation of algorithms on constraints by users. For deciding satisfiability of formulas, JConstraints translates from its internal object representation to the format expected by constraint solvers or a format suitable for different analysis goals. We demonstrate the capabilities of JConstraints by implementing a custom meta decision procedure for floating-point arithmetic that combines an approximating analysis over the reals with a proper floating-point analysis. The performance of the combined analysis is encouraging on a set of benchmarks: overall, a total reduction of time spent for constraint solving by 56% is achieved.

1 Introduction

Many problems in the analysis and formal verification of software can quite naturally be encoded as checking the satisfiability of a formula in some logic. Impressive advances in the past decades on the Boolean satisfiability problem and on satisfiability modulo theories (SMT) have made such encodings a viable approach in many cases. Today, there exists a plethora of tools and libraries that implement decision procedures with different profiles for a multitude of logics or fragments of logics [4].

For users of these decision procedures it is not easy to decide which implementation is particularly well-suited for a concrete analysis goal or problem instance. Moreover, most libraries have idiosyncratic native interfaces, making it hard to exchange one solver by another. The SMT-LIB Standard [3] mitigates this problem by defining a syntax for logic formulas that is supported by many solvers. A textual representation of formulas has some disadvantages, though: Encoded constraints are often specific to a domain: this can include structure of formulas, data types, and logic fragments. It is often beneficial to analyze, preprocess, and simplify constraints before submitting them to a constraint solver — as is, e.g., done by the Green tool, that implements canonization and caching on top of constraint solvers [34].

This paper, dedicated to Bernhard Steffen on the occasion of his 60th birthday, presents JConstraints, a Java library for working with logic constraints. JConstraints provides an object representation for logic expressions, native access to different SAT-, SMT- and interpolation-solvers, as well as useful tools and algorithms for working with logic constraints. The design philosophy behind JConstraints is heavily influenced by works of Bernhard Steffen:

- Object representation and programming interface borrow many concepts from the design of domain-specific languages [32]. Logic expressions can be represented at a level that is semantically close to the application domain, allowing, e.g., logic and arithmetic expressions with Java language types.
- Analysis of expressions with off-the-shelf constraint solvers is achieved by translating constraints to a representation suitable for analysis by a concrete constraint solver, following the design principle of the electronic tool integration (ETI) platform [33,24].

This enables JConstraints to maintain a representation and optimizations of constraints specific to the application domain (e.g., execution paths of Java programs) while utilizing the impressive performance and scalability of modern SMT solvers. It also allows the combination of different logic encodings and decision procedures. We demonstrate this feature by implementing a meta decision procedure, dubbed Feal, for floating-point arithmetic that combines decision procedures for floating-point arithmetic and reals as back-ends. The performance of the combined analysis is encouraging on a set of benchmarks: overall, a total reduction of time spent for constraint solving by 56% is achieved.

Outline. We provide an overview of JConstraints along with some examples of how the library can be used in Section 2. Section 3 details the meta decision procedure for floating-point arithmetic. Results from an evaluation on benchmarks from the literature are presented in Section 4. We discuss related work in Section 5 before concluding in Section 6.

2 The JConstraints Library

In this section, we first describe the architecture of JConstraints and then give a brief overview of the transformation of expressions into the input language of concrete constraint solvers. We also provide some code examples that illustrate the use of JConstraints. The code of JConstraints is published under the Apache License (Version 2.0) and is hosted on GitHub.[3] The library was first publicly released in 2015 and is used by a number of projects. We provide a short description of projects that depend on JConstraints for constraint representation and solver abstraction at the end of this section.

[3] `https://github.com/psycopaths/jconstraints`

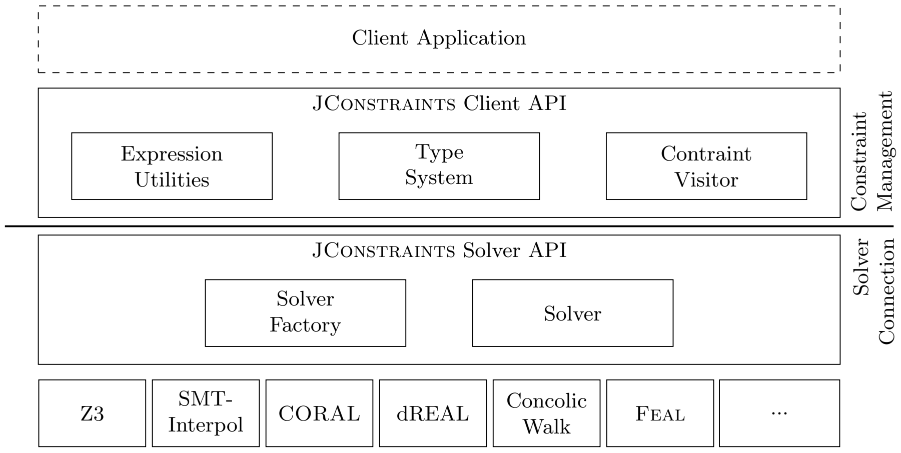

Fig. 1. The JConstraints architecture.

Architecture. Figure 1 sketches the architecture of JConstraints in detail: The upper part of the figure shows the client-side API, consisting of the basic library that provides an object representation for logic and arithmetic expressions and some basic utilities for working with expression objects (e.g., basic simplification, term replacement, and term evaluation). The object representation can be extended by custom types, and domain-specific algorithms on constraints that can be implemented using the visitor pattern. Utilities for building and reorganizing expressions are provided.

The lower part of the figure shows the solver-side API, consisting of solver factories for different solvers and a unified interface for interacting with different solvers. Plugins encapsulate the actual conversion from the general object representation into a solver input language and the necessary communication with the solver. Each of these plugins implements a solver factory and the unified solver interface for a specific constraint solver. Currently, JConstraints provides plugins for Z3 [12], SMTInterpol [8], CORAL [31], dReal [14] and the Concolic Walk algorithm [13]. The work described in Section 3 adds Feal, a meta-solver for floating-point problems.

Extending JConstraints by a new solver is very easy: It is sufficient to implement a factory that instantiates a solver (satisfying the unified solver interface). The solver object has to take care of translating to and from the new solver. Changing or selecting solver back-ends in client applications can be done through a configuration value either programatically or in a configuration file. Due to the separation into constraint representation layer and solver translation, benchmarking decision procedures of different solvers becomes exceptionally easy.

Type System and Evaluation. JConstraints manages variables as tuples of names and types. Out of the box, all Java types are supported. It is pos-

Listing 1.1. Example of JConstraints usage.

```
1  Variable x = create(BuiltinTypes.SINT32, "x");
2  Constant $c_5$ = create(BuiltinTypes.SINT32, 5);
3
4  Expression<Integer> $x_{plus_5}$ = plus(x, $c_5$);
5
6  Expression<Boolean> test = eq($x_{plus_5}$ , $c_5$);
7
8  Valuation val = new Valuation();
9  val.setValue(x, 0);
10
11 Integer i = $x_{plus_5}$.evaluate(val);
12 Boolean b = test.evaluate(val);
13
14 Properties conf = new Properties();
15 conf.setProperty("symbolic.dp", "z3");
16 ConstraintSolverFactory factory =
17   new ConstraintSolverFactory(conf);
18 ConstraintSolver solver = factory.createSolver();
19
20 Valuation model = new Valuation();
21 Result r = solver.solve(test, model);
22 assert r == Result.SAT;
```

sible to define application-specific types. The example in line 1 of Listing 1.1 demonstrates the declaration of a variable of signed integer type with 32 bit width. Similar to the type system of Java, the addition of two integers creates a new integer (see line 4 of the same listing). Boolean operations are modeled in an analogous fashion (line 6). Given a value assignment for all variables in an expression (valuation), the evaluation of the expression using Java semantics is possible (line 11). Finally, a constraint solver can be instantiated and called for deciding satisfiability of the expression.

Inside the call to the `solve()` method, types of variables are translated into some native type provided by some logic in constraint solver. Similar to the *object-relational impedance mismatch* (e.g. [28,20]) where semantic differences between Java's object types and SQL's relational type system occur, the conversion from Java types to the corresponding logic type system is not always straightforward. Different plugins may provide different mappings. In general, a mapping from Java types to SMT-LIB [3] theories and types is the first step. The result sometimes needs some tailoring for the specific solver. JConstraints aims at bridging this gap during SMT solver integration, so that a comparable development experience with using ORM system for database access is established. In case a solver supports the generation of models, the resulting valuation will be extracted for satisfiable expressions and transformed back to Java types (line 18 and line 21) of Listing 1.1. This model might be used to verify the solvers verdict using the evaluation functionality discussed above.

Instead of using JConstraints, the problem from Listing 1.1 might have been expressed in the SMT-LIB constraint language, which is the standardized description language for SMT problems supported by many SMT-solvers. Listing 1.3 demonstrates how the constraints from Listing 1.1 might be described in

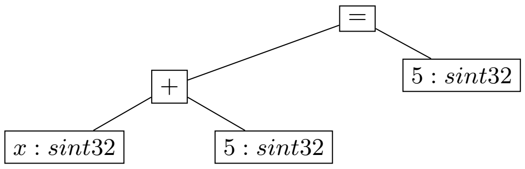

Fig. 2. Object representation of the expression from Listing 1.1.

the SMT-LIB language. When integrating SMT-LIB into JAVA, the same difficulties arise that have been described in the past for the integration of SQL [28]: The resulting JAVA code consists of String concatination instead of describing the real problem, which is now in the String content. Moreover, SMT-LIB uses prefix notation, while JAVA and most of handwritten problems are expressed in infix notation. As a consequence, transforming a problem into SMT-LIB format requires a mental shift during programming, mixing these two notation formats. JCONSTRAINTS supports parsing constraints from strings in infix notation avoding this problem. The constraint from Figure 2 could be parsed by JCONSTRAINTS from the string $(('x' : sint32\ +\ 5)\ ==\ 5)$.

Constraint Representation. Constraints are represented as trees of objects in JCONSTRAINTS. Figure 2 sketches the tree created for the expression from Listing 1.1: The expression `test` from line 6 decomposes into an addition on the left side of the expression and a single constant on the right side.

The constraint manipulation API of JCONSTRAINTS provides a set of default constraint visitors for traversing the tree representation. There is, e.g., a duplicating visitor, that duplicates a complete expression while traversing the tree. A visitor for renaming variables can easily be implemented by extending the behavior of the duplicating visitor to rename variables before cloning. Listing 1.2 demonstrates the code required. The method definition in line 5 overwrites the behavior of the duplicating visitor for nodes in the tree that represent variables.

Listing 1.2. Renaming Visitor.

```
1   class RenameVarVisitor extends
2       DuplicatingVisitor<Map<String, String>> {
3
4     @Override
5     public <E> Expression<?> visit(Variable<E> v,
6         Map<String, String> data) {
7       String newName = data.get(v.getName());
8       return Variable.create(v.getType(), newName);
9     }
10
11    public <T> Expression<T> rename(Expression<T> expr,
12        Map<String,String> renaming) {
13      return visit(expr, renaming).requireAs(expr.getType());
14    }
15  }
```

Listing 1.3. Example of SMT-LIB usage.

```
...
String smtProblem = "(declare-fun x () Int)\n";
smtProblem += "(declare-fun tmp1 () Int)\n";
smtProblem += "(assert (= tmp1 (+ x 5)))\n";
smtProblem += "(assert (= tmp1 5))\n";
smtProblem += "(assert (= x 0))\n";
smtProblem += "(check-sat)\n";
...
```

Listing 1.4. Translation to Z3.

```
1   @Override
2   public Expr visit(PropositionalCompound n, Void data) {
3     BoolExpr left = null, right = null;
4     try {
5       left = (BoolExpr)visit(n.getLeft(), null);
6       right = (BoolExpr)visit(n.getRight(), null);
7
8       switch(n.getOperator()) {
9       case AND:
10        return ctx.mkAnd(left, right);
11      case OR:
12        return ctx.mkOr(left, right);
13      ...
14    }
15    catch(Z3Exception ex) {...}
16  }
```

On the basis of these easily extensible visitors many analyses and transformation tasks can be broken down into local operations on tree nodes.

Constraint Solving. As discussed above, different constraint solvers can be used in JConstraints as plugins. Translation from JConstraints's object representation to solver-specific representations of constraints is based on the visitor pattern, too. Listing 1.4 shows an excerpt of the visitor that translates to Z3. The method in the listing handles object of type `PropositionalCompund` which is JConstraints's expression sub-type for logical compounds (e.g., conjunctions). In this case, when encountering a conjunction in JConstraints, a corresponding conjunction is created using Z3's native interface in line 10 of Listing 1.4. The two conjuncts `left` and `right` have already corresponding native Z3 representations (data type *BoolExpr*) in line 5 and line 6.

For most of the supported solvers JConstraints currently relies on provided JNI interfaces. The dReal solver is an exception: it is connected using a visitor that translates to a subset of the SMT-LIB language.

Applications. JConstraints is used in different applications. Historically, it has been developed along with JDart [23].

JDart. JDart is a concolic execution engine for Java based on JPF that can be used for generating test cases as well as the symbolic summaries for methods.

The tool executes Java programs with concrete and symbolic values at the same time and records symbolic constraints describing all the decisions along a particular path of the execution. These path constraints are then used to find new paths in the program. Concrete data values for exercising these paths are generated using a constraint solver. JConstraints has been the central constraint management library and solver connector in this project.

RaLib. RaLib [7], an extension of LearnLib [21] for learning register automata [19], uses JConstraints for representing transition guards of extended finite state machines and for finding concrete data values for executing sequences of guarded transitions.

Psyco. The Psyco tool [16] generates and verifies symbolic behavioral interfaces for software components using a combination of multiple dynamic and static analysis techniques: active automata learning, concolic execution, static code analysis, symbolic search, predicate abstraction, and model-based testing. Especially the concolic execution, symbolic search and predicate abstraction modules depend on JConstraints for representing constraints.

3 Feal: Multi-Theory Solving for Floats

In this section we describe how JConstraints can be used for defining meta-constraint solvers by integrating multiple decision procedures as back-ends. We demonstrate this capability by presenting a meta-analysis for floating-point expressions for which we combine an encoding and analysis over reals with one based on floating-point numbers. As we will show in the next section, the resulting meta-analysis improves the efficiency of dynamic symbolic execution over floating-point computations on a set of standard benchmarks.

Our meta-analysis is based on the observation that solving floating-point constraints after approximating them over reals is often more efficient than solving them over floating-point arithmetic. Such an analysis will find models in many cases. In some cases the models can be spurious (i.e., not be models when translated to floating point representation). Moreover, unsatisfiability verdicts may be spurious, too: e.g., due to rounding behavior the expression $a + b = a \wedge a \neq 0$ is satisfiable over floating point variables but trivially unsatisfiable over reals. Using an encoding over reals can serve as a fast generator for candidate models and verdicts, which then have to be validated using evaluation or analysis in floating point semantics.

The correctness of candidate models can be checked on the JConstraints representation by evaluating obtained models on constraints using the actual Java floating-point implementation. In case a solution is correct (i.e., a model in floating-point semantics), we can save a call to a usually much more expensive floating-point decision procedure. This in turn reduces the total solving time and improves the run time of applications. In case a solution does not satisfy the constraints in floating-point arithmetic, we can still resort to a floating-point decision procedure.

On the other hand, if the decision procedure over reals concludes that a set of constraints is not satisfiable, this verdict has to be substantiated by a

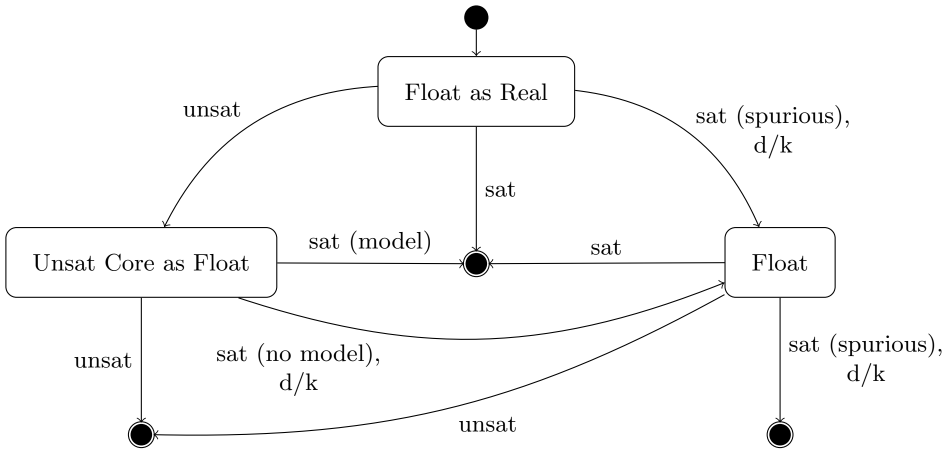

Fig. 3. Control-flow in the meta-analysis that combines solving over reals and solving in floating-point arithmetic.

floating-point decision procedure (cf. above example). We can, however, leverage the unsafisfiable core, i.e., a smallest unsatisfiable subset of the analyzed constraints, as a heuristic for optimizing the floating-point analysis: if this core proves unsatisfiable in floating-point arithmetic, we can circumvent analyzing the (possibly much larger) complete set of constraints in floating-point arithmetic.

The complete control-flow of the combined decision procedure is shown in Figure 3. It can be divided into three stages:

Real: Floating-point constraints are approximated as constraints over rational numbers. A constraint solver that supports the real arithmetic theory is then used to check whether the resulting constraints are satisfiable. If this is the case, the solver will generate a model that satisfies the constraints over the reals. If the model can be validated over floating-point arithmetic (as described above), the decision procedure terminates with this model (sat). In case the model cannot be validated (spurious sat), or if the real analysis terminates inconclusively (dont know) we resort to a floating-point analysis in stage *Float*. If the constraints are found to be unsatisfiable by the real solver, we proceed to analyze the unsatisfiable core in stage *UnsatCore*.

UnsatCore: If the constraints are found to be unsatisfiable by the real analysis, an unsatisfiable core is obtained from the constraint solver. This core contains a subset of the original constraints, such that the conjunction of the constraints in this subset is still unsatisfiable. Satisfiability of this core is then checked using a floating-point decision procedure. If the core is unsatisfiable in floating-point arithmetic, the result of the first stage is substantiated and the analysis concludes (unsat). In case the core is satisfiable in floating-point arithmetic, a model can be obtained from the constraint solver. If this model of the core can be extended to a model that satisfies the complete set of constraints, this model is returned (sat model) and the analysis terminates. If analysis on the core terminates inconclusively (dont know) or the obtained model does not satisfy the complete set of constraints (sat no model), we resort to a floating-point analysis in stage *Float*.

Float: In the final stage, the complete set of constraints is analyzed without any approximation by a solver that supports floating-point arithmetic. The analysis may find a model (sat), determine unsatisfiability (unsat), or terminate inconclusively (dont know). In case of different floating-point semantics in the constraint solver and the target language (Java), the analysis may conclude that a model is spurious (sat spurious).

In the sketched analysis, inconclusive termination of constraint solvers may arise due to timeouts, insufficient solver capabilities, or constraints that are computationally intractable.

It is easy to see that some of the paths shown in the control-flow diagram in Figure 3 are more expensive than using a floating-point decision procedure directly. Hence, instead of using the floating-point analysis only as a last resort, we simply run the multi-step analysis sketched above in parallel with a floating-point analysis in our implementation. Whichever analysis terminates first determines result and analysis time of the meta-analysis.

The domain-specific type system of JConstraints and the compilation-based approach to integrating constraint solvers in JConstraints makes it easy to implement the necessary sequence of analysis steps and transformations as well as validation of obtained models in Java.

4 Experimental Evaluation

In this section we report the results of an experimental evaluation of the approach discussed in the previous section. We aim at evaluating the effectiveness and efficiency of the presented analysis.

Experimental Setup. We base our evaluation on the analysis of path constraints in the dynamic symbolic analysis framework JDart [23]. We compare the time spent on constraint solving when using our meta-analysis to the time spent when using a floating-point decision procedure. JDart is a suitable driver for our experiments as it uses JConstraints internally and allows us to replace only the constraint solver while using identical setups otherwise. For every symbolic path in the analyzed program, we record the time spent for constraint solving and compare the two evaluated approaches. All experiments were performed on a laptop with Intel Core i5 2.3 GHz processor and 8 GB of RAM running macOS 10.13.2. For constrains solving Microsoft Z3 [12] version 4.6.0 was used with a timeout of 10 second per solver invocation.

Table 1. Performance Improvement of Feal over Floating-Point by Constraint Solver Verdict.

Verdict	Instances [#]	Feal/ FP per Path [%]	WCT FP [sec]	WCT Feal [sec]	Optimized Paths [%]
unsat	358.00 (6.71)	72.22 (38.21)	3,843.65 (5,378.10)	1,202.81 (783.17)	55.49 (1.14)
sat	176.60 (4.16)	15.87 (8.82)	549.34 (515.09)	52.17 (17.54)	80.30 (5.85)
d/k	41.00 (1.73)	100.00 (0.00)	766.24 (1,294.46)	766.24 (1,294.46)	0.00 (0.00)
sat(sp.)	15.00 (2.24)	100.00 (0.00)	428.15 (928.08)	428.15 (928.08)	0.00 (0.00)

Selection of Benchmarks. A sample of benchmark functions was selected from multiple sources including open source projects on GitHub and other publications that focus on analysis of floating-point computations in programs [30,26,2]. In order to include some safety-critical methods in our analysis, we rewrote some C++ embedded software functions in Java (e.g., from TCAS [23]). Since JDart analyses methods for symbolic method parameters, we use only Java methods as benchmarks that contain floating-point computations depending on method parameters. We use a total of 15 benchmark methods.

Since symbolic execution engines for java programs are usually incapable of analyzing native implementation of mathematical functions (e.g., sqrt, sin, etc.), we replaced all calls to elementary mathematical functions in the benchmarks with calls to simpler versions written in Java that can be completely analyzed.

Experimental Results. Since the approach from Section 3 is designed to perform better or (in the worst case) only as good as a floating-point analysis, we report improvement (in runtime) over this baseline. We report averages and standard deviations from five runs for each benchmark.

On average, the cumulated wall clock time for all benchmarks was reduced by 56% from $5,587$ seconds (std. dev. $7,428$) to $2,449$ seconds (std. dev. $2,076$). For individual paths in programs (i.e., calls to the constraint solver) solving time was reduced by 28% on average (std. dev. 28%) and $57,65\%$ of paths (with a very low standard deviation of $1,53\%$) were solved faster by Feal than by the floating-point decision procedure.

Table 1 reports number of instances, average improvement over the baseline for individual paths, wall clock times for floating-point analysis and Feal, as well as the average percentage of paths on which solving could be optimized for different possible verdicts of the decision procedure. Standard deviations are reported in parentheses. As the data shows, we observe improvements for constraint sets that are found satisfiable (84.13% reduction of runtime on average per path and 90.50% reduction of wall clock time) or unsatisfiable (27.78% reduction of runtime on average per path and 68.71% reduction of wall clock time). Together these cases account for 90.51% of analyzed path constraints and 78.62% of required runtime.

Table 2 shows a more detailed analysis for individual benchmarks. It can be observed that Feal is effective for every analyzed method. Performance gains

Table 2. Performance Improvement of FEAL over Floating-Point by Benchmark.

Benchmark	Paths [#]	WCT FP [sec]	WCT FEAL [sec]	Amenable Paths [%]
generate_star	52.00 (0.00)	2,593.12 (5,648.60)	37.52 (1.67)	88,46 (0,00)
rgb_to_hsl	70.00 (0.00)	934.99 (1,292.98)	840.83 (1,291.55)	44,86 (2,17)
sv_newton	2.00 (0.00)	909.21 (769.34)	896.39 (782.15)	20,00 (44,72)
hsl_to_rgb	127.00 (0.00)	458.29 (929.61)	438.38 (928.48)	67,72 (0,79)
sm_sin	28.00 (2.24)	261.25 (536.32)	17.24 (0.92)	36,78 (5,87)
sv_arctan_Pade_true	12.00 (0.00)	203.79 (168.61)	93.67 (16.48)	56,67 (6,97)
getClosestPointOnSeg	19.00 (0.00)	104.96 (1.96)	25.57 (3.23)	62,11 (2,35)
interpolate_color	4.00 (0.00)	51.42 (1.76)	46.23 (1.52)	25,00 (0,00)
get_x_y	5.00 (0.00)	33.12 (3.35)	26.69 (15.24)	36,00 (35,78)
sv_exp_loop_true	200.20 (2.68)	17.53 (1.17)	11.52 (0.48)	53,51 (3,17)
e_adventure	6.00 (0.00)	11.56 (0.30)	10.01 (0.25)	33,33 (0,00)
sm_exp	38.40 (5.37)	5.13 (0.65)	3.87 (0.49)	32,50 (1,86)
sv_squer_2_var	3.00 (0.00)	2.24 (0.13)	1.34 (0.39)	40,00 (14,91)
sm_rint	10.00 (0.00)	0.59 (0.04)	0.07 (0.03)	98,00 (4,47)
sm_atan	14.00 (0.00)	0.19 (0.01)	0.04 (0.02)	88,57 (9,58)

range from 1.41% (for *sv_newton*) to 98.55% (for *generate_star*) reduction in wall clock time. All methods have a significant number of paths on which FEAL outperforms the floating-point analysis. As the reported standard deviations indicate, some paths are not consistently solved more efficiently by FEAL.

Table 3 shows performance improvements on different paths of the control-flow diagram shown in Figure 3. Unsurprisingly, performance improvements can be observed in cases that only use stages *Real* and *UnsatCore* (79.31% of tree traversals). As the data shows, all control-flow paths are exercised by the benchmarks.

Threats to Validity. Generalizability of the observed performance improvements may be limited by two aspects of our experimental setup. First, our analysis can only show that there exists a potential for improvement for the sampled methods. We cannot estimate how representative our findings are for the set of all methods that contain floating-point computations. A second threat to the validity of the reported findings arises from the fact we replaced all calls to elementary mathematical functions with calls to versions written in JAVA. As a consequence, our analysis does not allow us to draw consequences about analyzing JAVA programs with floating-point computations in general.

5 Related Work

Constraint Libraries. There are several libraries for representing logic constraints and for providing a unified interface to constraint solvers. For JAVA, JAVASMT [4] provides constraint representation and unified access to multiple

[4] `https://github.com/sosy-lab/java-smt`

Table 3. Performance Improvement of FEAL over Floating-Point for Stages and Verdicts.

Stages	Verdict	Instances [#]	FEAL/ FP WCT [%]
Real	sat	142.60 (14.22)	1.33 (0.70)
Real, UnsatCore	unsat	325.80 (7.98)	56.66 (33.69)
Real, UnsatCore	sat	1.00 (0.00)	100.00 (0.00)
Real, Float	unsat	22.00 (4.47)	100.00 (0.00)
Real, Float	sat	32.00 (10.07)	100.00 (0.00)
Real, Float	d/k	38.20 (4.60)	100.00 (0.00)
Real, Float	sat (spurious)	8.00 (2.24)	100.00 (0.00)
Real, UnsatCore, Float	unsat	10.20 (3.35)	100.00 (0.00)
Real, UnsatCore, Float	sat	1.00 (0.00)	100.00 (0.00)
Real, UnsatCore, Float	d/k	2.80 (6.26)	100.00 (0.00)
Real, UnsatCore, Float	sat (spurious)	7.00 (0.00)	100.00 (0.00)

SMT solvers. The type system of JAVASMT mirrors types of the logics supported by solvers. JCONSTRAINTS, in contrast provides a flexible type system that allows to model types of target domains. For Python, PYSMT [15] provides constraint representation and constraint solver abstraction. PYSMT provides so-called *portfolio solving* (running multiple constraint solvers in parallel) but not the combination of solvers in more complex patterns.

Analysis of Floating-Point Computations. The verification of floating point operations has a long history. First approaches, such as the one described by Aharoni et al. [1] used a combination of floating point constraints solving engines to solve data constraints on operands of individual instructions to generate test suites. Those suites aimed at the corner cases of floating-point operations for bug discovery. More recent test generators use symbolic execution to cover all possible execution branches and their intervals and use the symbolic execution output as input for the test case generation of the numeric floating point functions. Schumann et al. [29] build such a tool based on the KLEE tool [6] and generated test cases for an open source autopilot[5]. Liew et al. [22] extended KLEE towards a complete symbolic execution engine for floating-point programs based on constraint solving. This was achieved via using an of the shelf SMT solver that supports floating-point reasoning. Another symbolic execution engine for floating point C-programs that bundles various contribution in this area is FPSE first introduced by Botella et al. [5]. Their symbolic execution engine used a solver based on floating point interval propagation that is dedicated for floating point number.

Various abstract interpretation based methods have been proposed for verifying floating-point properties. Many of them focus the precision of the floating-

[5] https://github.com/ArduPilot/ardupilot

point computations rather than potential run-time errors. Martel [25] introduced a concrete semantic for the propagation of round-off errors throughout the floating-point computations expressed in first order terms. Solving the combination of first order terms allows checking of eventual introduction of round-off errors. Gouboult and Putot implemented the tool FLUCTUAT [17,27,18] which uses abstract semantics and abstract domain for the static analysis of floating-point computations. Their approach is based on domains for bounding the ranges of the floating-point variables. These domains allow to analyze potential round-off errors during the floating-point computations.

Integrated Formal Analysis Methods. Though not directly related to the work we present here at a technical level there is a number of works that propose integration of multiple formal methods with different profiles with the purpose of optimizing effectiveness and efficiency: In context of analyzing software product lines, Damiani et al. present a meta decision procedure that decomposes the analyzed problem and computes results for sub-problems using multiple analysis methods with different profiles [10]. Cousot et. al. [9] combine different abstract domains during abstract interpretation and the work of Darulova and Kuncak [11] combines an exact SMT solver with affine and interval arithmetics.

6 Conclusion

In this paper, dedicated to Bernhard Steffen on the occasion of his 60th birthday, we have presented JConstraints, a constraint solver abstraction layer for Java. JConstraints provides an object representation for logic expressions, unified access to different SMT and interpolation solvers, and useful tools and algorithms for working with logic formulas. The design philosophy behind JConstraints is heavily influenced by works of Bernhard Steffen: Object representation and programming interface borrow many concepts from the design of domain-specific languages [32]. Logic expressions can be represented at a level that is semantically close to the application domain, abstracting from the encoding constraint solvers support. Analysis of expressions with off-the-shelf constraint solvers is achieved by translating constraints to a representation suitable for analysis by a concrete constraint solver, following the design principle of the electronic tool integration (ETI) platform [33,24], allowing to generate and analyze views on a problem by translating to different back-ends.

We have demonstrated the capabilities of JConstraints by implementing a custom meta decision procedure for floating-point arithmetic that combines an approximating analysis over the reals with a proper floating-point analysis. In a small evaluation in the context of symbolic execution, the meta decision procedure reduces time spent for constraint solving by 56%. As a next step, we plan to implement and evaluate several other meta decision procedures combining solvers with different profiles, e.g., one that does not provide models with one that does searches for models but cannot decide unsatisfiability.

Acknowledgments. JConstraints has been developed as part of JDart and several people who worked on JDart have contributed ideas and code to JConstraints. Notable contributions to JConstraints have been made by Marko Dimjašević, Malte Isberner, and Kasper Luckow.

References

1. Merav Aharoni, Sigal Asaf, Laurent Fournier, Anatoly Koifman, and Raviv Nagel. Fpgen-a test generation framework for datapath floating-point verification. In *High-Level Design Validation and Test Workshop, 2003. Eighth IEEE International*, pages 17–22. IEEE, 2003.
2. Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. Symbolic path-oriented test data generation for floating-point programs. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 1–10. IEEE, 2013.
3. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.
4. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
5. Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2):97–121, 2006.
6. Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
7. Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning extended finite state machines. In *International Conference on Software Engineering and Formal Methods*, pages 250–264. Springer, 2014.
8. Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. Smtinterpol: An interpolating SMT solver. In *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, pages 248–254, 2012.
9. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In *Annual Asian Computing Science Conference*, pages 272–300. Springer, 2006.
10. Ferruccio Damiani, Reiner Hähnle, and Michael Lienhardt. Abstraction refinement for the analysis of software product lines. In *Tests and Proofs - 11th International Conference, TAP 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings*, pages 3–20, 2017.
11. Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *Acm Sigplan Notices*, volume 49, pages 235–248. ACM, 2014.
12. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
13. Peter Dinges and Gul Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the 22nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, Hong Kong, November 16-21 2014. ACM.

14. Sicun Gao, Soonho Kong, and Edmund M Clarke. dreal: An smt solver for nonlinear theories over the reals. In *International Conference on Automated Deduction*, pages 208–214. Springer, 2013.
15. Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*, 2015.
16. Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 248–264, Berlin, Heidelberg, 2012. Springer-Verlag.
17. Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *International Static Analysis Symposium*, pages 18–34. Springer, 2006.
18. Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 232–247. Springer, 2011.
19. Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 251–266, 2012.
20. Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pages 36–43. IEEE, 2009.
21. Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib - A framework for active automata learning. In *CAV 2015, Part I*, pages 487–495, 2015. (Best Artifact Award).
22. Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zahl, and Klaus Wehrle. Floating-point symbolic execution: A case study in n-version programming. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*, pages 601–612. IEEE, 2017.
23. Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. Jdart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459. Springer, 2016.
24. Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. Remote integration and coordination of verification tools in JETI. In *12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), 4-7 April 2005, Greenbelt, MD, USA*, pages 431–436, 2005.
25. Matthieu Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *European Symposium on Programming*, pages 194–208. Springer, 2002.
26. Corina Pasareanu, Marcelo dAmorim, Mateus Borges, and Matheus Souza. Coral: Solving complex constraints for symbolic pathfinder. 2010.
27. Sylvie Putot, Eric Goubault, and Matthieu Martel. Static analysis-based validation of floating-point computations. In *Numerical software with result verification*, pages 306–313. Springer, 2004.
28. Keven Richly, Martin Lorenz, and Sebastian Oergel. S4j-integrating sql into java at compiler-level. In *International Conference on Information and Software Technologies*, pages 300–315. Springer, 2016.

29. Johann Schumann and Stefan-Alexander Schneider. Automated testcase generation for numerical support functions in embedded systems. In *NASA Formal Methods Symposium*, pages 252–257. Springer, 2014.
30. Elena Sherman and Matthew B Dwyer. Exploiting domain and program structure to synthesize efficient and precise data flow analyses (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 608–618. IEEE, 2015.
31. Matheus Souza, Mateus Borges, Marcelo dAmorim, and Corina S Păsăreanu. Coral: solving complex constraints for symbolic pathfinder. In *NASA Formal Methods Symposium*, pages 359–374. Springer, 2011.
32. Bernhard Steffen, Frederik Gossen, Stefan Naujokat, and Tiziana Margaria. Language-driven engineering: From general purpose to purpose-specific languages. *LNCS*, 10000, to appear.
33. Bernhard Steffen, Tiziana Margaria, and Volker Braun. The electronic tool integration platform: Concepts and design. *STTT*, 1(1-2):9–30, 1997.
34. Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 58, 2012.