FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution

Kiran Lakhotia[1], Nikolai Tillmann[2], Mark Harman[1], and Jonathan de Halleux[2]

[1] University College London, CREST Centre, Department of Computer Science, Malet Place, London, WC1E 6BT, UK.
[2] Microsoft Research, One Microsoft Way, Redmond WA 98052, USA

Abstract. Recently there has been an upsurge of interest in both, Search–Based Software Testing (SBST), and Dynamic Symbolic Execution (DSE). Each of these two approaches has complementary strengths and weaknesses, making it a natural choice to explore the degree to which the strengths of one can be exploited to offset the weakness of the other. This paper introduces an augmented version of DSE that uses a SBST–based approach to handling floating point computations, which are known to be problematic for vanilla DSE. The approach has been implemented as a plug in for the Microsoft Pex DSE testing tool. The paper presents results from both, standard evaluation benchmarks, and two open source programs.

1 Introduction

It is widely believed that automating parts of the testing process is essential for successful and cost effective testing. Of the many approaches to automated test data generation in the literature, two important and widely studied schools of thought can be found in the work on Search–Based Software Testing (SBST) and Dynamic Symbolic Execution (DSE). This paper proposes an augmented approach to tackle the problem of floating point computations in DSE. It has been implemented as an extension to the Microsoft DSE testing tool Pex.

SBST and DSE have different strength and weaknesses. For instance, SBST provides a natural way to express coverage based test adequacy criteria as a multi–objective problem [6], while DSE is better suited to discover the structure of the input to the system under test. Yet there has been little work in trying to combine SBST and DSE.

Inkumsah and Xie [8] were the first authors to propose a framework (EVACON) combining evolutionary testing with DSE. Their framework targets test data generation for object oriented code written in JAVA. They use two existing tools, eToc [18], an evolutionary test data generation tool, and jCUTE [16], a DSE tool. eToc constructs method sequences to put a class containing the method under test, as well as non–primitive arguments, into specific desirable states. jCUTE is then used to explore the state space around the points reached by eToc.

Majumdar and Sen [10] combined random testing with DSE in order to address the state problem in unit testing. A random search is used to explore

the state space, while DSE is used to provide a locally exhaustive exploration from the points reached by the random search.

Neither of the two approaches addresses the issue of floating point computations in DSE, which originate from limitations of most constraint solvers. Constraint solvers approximate constraints over floating point numbers as constraints over rational numbers. Because computers only have limited precision, solutions which are valid for rational numbers are not always valid when mapped to floating point numbers. Thus, Botella et al. [2] proposed a constraint solver based on interval analysis in order to symbolically execute programs with floating point computations. A limitation of their work is that it does not consider combinations of integers and floating point expressions.

The approach proposed in this paper differs from previous work in that it provides a general framework for handling constraints over floating point variables. It is also the first paper to propose a combination of DSE with SBST in order to improve code coverage in the presence of floating point computations.

The rest of the paper is organized as follows: Section 2 presents a brief overview of SBST and DSE, and the two search algorithms implemented. Section 3 describes the DSE tool Pex and how it has been extended with arithmetic solvers for floating point computations. The empirical study used to evaluate the solvers, analysis of the results, and threats to validity are presented in Section 4. Section 5 concludes.

2 Background

2.1 Dynamic Symbolic Execution

In DSE [5, 17] the goal is to explore and execute the program to achieve structural coverage, based on analysis of the paths covered thus far. DSE combines symbolic and concrete execution. Concrete execution drives the symbolic exploration of a program, and runtime values can be used to simplify path constraints produced by symbolic execution to make them more amenable to constraint solving. Consider the example shown in Figure 1 and suppose the function is executed with the inputs `a=38` and `b=100`. The execution follows the `else`-branch at the `if`-statement. The corresponding path condition is $(int)\ Math.Log\ (a_0) \neq b_0$, where a_0 and b_0 refer to the symbolic values of the input variables `a` and `b` respectively. In order to explore a new execution path, the path condition is modified, for example by inverting the last constraint, *i.e.* $(int)\ Math.Log\ (a_0) = b_0$. However, suppose that the expression `(int)Math.Log(a)` cannot be handled by a particular constraint solver. In order to continue testing, DSE replaces the expression `(int)Math.Log(a)` with its runtime value, for example 3. The path condition can thus be simplified to $3 == b_0$, which can now be solved for b_0. Executing the program with the (updated) input values `a=38` and `b=3` traverses the `then`-branch of the `if`-statement.

2.2 Search–Based Testing

The field of SBST began in 1976 with the work of Miller and Spooner [11], who applied numerical optimization techniques to generate test data which traverses

```
void testme2(double a, int b)
{
   if( (int)Math.Log(a) == b )
      //
}
```

Fig. 1. Example used for demonstrating dynamic symbolic execution and search–based testing.

a selected path through the program. In SBST, an optimization algorithm is guided by an objective function to generate test data. The objective function is defined in terms of a test adequacy criterion. Recall the example from Figure 1 and suppose the condition in the `if`-statement must be executed as true. A possible objective function is $|(int)Math.Log(a) - b|$. When this function is 0, the desired input values have been found. Different functions exist for various relational operators in predicates [9] and objective functions have been developed to generate test data for a variety of program structures, including branches and paths [19].

2.3 Alternating Variable Method

A simple but effective optimization technique [7], known as the Alternating Variable Method (AVM), was introduced by Korel [9] in the 1990s. It is a form of hill climbing and works by continuously changing an input parameter to a function in isolation. Initially all (arithmetic type) inputs are initialized with random values. Then, so called *exploratory moves* are made for each input in turn. These consist of adding or subtracting a delta from the value of an input. For integral types the delta starts off at 1, *i.e.*, the smallest increment (decrement). When a change leads to an improved fitness value, the search tries to accelerate towards an optimum by increasing the size of the neighborhood move with every step. These are known as *pattern moves*. The formula used to calculate the delta added or subtracted from an input is: $\delta = 2^{it} * dir * 10^{-prec_i}$, where it is the repeat iteration of the current move (for pattern moves), dir either -1 or 1, and $prec_i$ the precision of the ith input variable. The precision applies to floating point variables only (*i.e.* it is 0 for integral types). It denotes a scale factor for the size of a neighborhood move. For example, setting the precision ($prec_i$) of an input to 1 limits the smallest possible move to ± 0.1. Increasing the precision to 2 limits the smallest possible move to ± 0.01, and so forth.

Once no further improvements can be found for an input, the search continues optimizing the next input parameter, and may recommence with the first input if necessary. In case the search stagnates, *i.e.* no move leads to an improvement, the search restarts at another randomly chosen location in the search–space. This is known as a *random restart* strategy and is designed to overcome local optima and enable the AVM to explore a wider region of the input domain for the function under test.

2.4 Evolution Strategies

Evolution Strategies (ES) date back to the 1960s and were popularized in the early '70s by Rechenberg [14] and Schwefel [15]. They belong to the family of Evolutionary Algorithms (EA). In an EA, a population of individuals *evolves* over a number of generations. Between each generation, genetic operators are applied to the individuals, loosely representing the effects of mating (crossover) and mutation in natural genetics. The net effect of these operations is that the population becomes increasingly dominated by better (more fit) individuals.

In an ES, an individual has at least two components: an object vector (*i.e.* containing the inputs to the function under test) and a strategy parameter vector. The strategy parameters 'evolve' themselves as the search progresses. Their role is to control the strength of the mutations of the object vector, so that the ES can self–adapt to the underlying search landscape. For example, if the search is far away from a global optimum, large mutations enable it to make faster progress towards an optimum. Conversely, the closer the search is to an optimum, smaller mutations may be necessary to reach the optimum.

3 Implementation

We will now describe how the AVM from Section 2.3 and the ES from Section 2.4 have been implemented as a Pex extension called FloPSy (search-based **Flo**ating **P**oint constraint solving for **Sy**mbolic execution). The extension is open source and available online from the codeplex website; URL: http://pexarithmeticsolver.codeplex.com.

3.1 Pex

Pex [17] is a test input generator for .NET code, based on DSE; test inputs are generated for parameterized unit tests, or for arbitrary methods of the code under test.

Pex can symbolically represent all primitive values and operations, including floating point operations and calls to pure functions of the .NET `Math` class. Note that unlike traditional DSE, Pex only simplifies constraint systems by using concrete values when the code under test invokes an environment-facing method, *i.e.* a method that is defined outside of .NET, and not part of the `Math` class.

Pex implements various heuristics to select execution paths. The constraint solver Z3 [3] is used to decide the feasibility of individual execution paths. Z3 is a Satisfiability Modulo Theories (SMT) solver, which combines decision procedures for several theories. However, Z3 does not have a decision procedure for floating point arithmetic. When Pex constructs constraint systems for Z3, all floating point values are approximated by rational numbers, and most operations are encoded as uninterpreted functions. Pex supports *custom arithmetic solvers* to work around this restriction.

Path selection and path constraint solving: Given a program with input parameters, Pex generates test inputs in an iterative fashion. At every point

in time, Pex maintains a representation of the already explored execution tree defined by the already explored execution paths of the program. The branches of this tree are annotated by the branch conditions (expressed over the input parameters). In every step of the test input generation, Pex selects an already explored path prefix from the tree, such that, not all outgoing branches of the last node of the path prefix have been considered yet. An outgoing branch has been considered when it has been observed as part of a concrete execution path, when it has been classed as infeasible, or when constraint solving timed out. Pex then constructs a constraint system that represents the condition under which the path prefix is exercised, conjoined with the negation of the disjunction of all outgoing branch conditions. If a solution of this constraint system can be obtained, then we have found a new test input which – by construction – will exercise a new execution path which was not part of the already explored execution tree. The path is then incorporated into the tree.

The successive selection of path prefixes plays a crucial role for the effectiveness of test generation. If the program contains unbounded loops or recursion, the execution tree might be infinite, and thus the selection of path prefixes must be fair in order to eventually cover all paths. Even if loops and recursion are bounded, or if the program does not contain loops and recursion, a fair selection is still important to achieve high code coverage fast. Pex implements various heuristic strategies to select the next path prefix [21], incorporating code coverage metrics and fitness functions.

Each constraint system is a Boolean-valued expression over the test inputs. Pex' expression constructors include primitive constants for all basic .NET data types (integers, floating point numbers, object references), and functions over those basic types representing particular machine instructions, *e.g.* addition and multiplication, and the functions of the .NET `Math` class. Pex uses tuples to represent .NET value types ("structs") as well as indices of multi-dimensional arrays, and maps to represent instance fields and arrays, similar to the heap encoding of ESC/Java [4]: An instance field of an object is represented by a *field map* which associates object references with field values. For each declared field in the program, there is one location in the state that holds the current field map value. An array type is represented by a class with two fields: a length field, and a field that holds a mapping from integers (or tuples of integers for multi-dimensional arrays) to the array elements. Constraints over the .NET type system and virtual method dispatch lookups are encoded in expressions as well. Most of Pex' expressions can be mapped directly into SMT theories for which Z3 has built-in decision procedures, in particular propositional logic, fixed sized bit-vectors, tuples, arrays, and quantifiers. Floating point values are functions that are a notable exception.

Extension mechanism for floating point constraint solving: If some constraints refer to floating point operations, then Pex performs a two-phase solving approach: First, all floating point values are approximated by rational numbers, and most operations are encoded as uninterpreted functions, and it is checked whether the resulting constraint system is satisfiable; if so, a model, *i.e.* a sat-

isfying assignment, is obtained. (Note that this model includes a model for the uninterpreted functions, which might not reflect the actual floating point semantics.) Second, one or more custom arithmetic solvers are invoked in order to correct the previously computed model at all positions which depend on floating point constraints.

Consider for example the following method.

```
void foo(double[] a, int i){
  if (i >= 0 && i < a.Length &&
      Math.Sin(a[i]) > 0.0) {
    ...
  }
}
```

In order to enter the `then`-branch of the `if`-statement, the three conjuncts must be fulfilled. Using Z3, we may obtain a model, *i.e.* a satisfying assignment, where $i = 0$ and $a.\texttt{Length} = 1$ and $a[i] = 0/1$ (0 or 1) and $\texttt{Math.Sin}(0/1) = 1/1$ is a solution of the constraint system, when approximating floating point values with rational numbers, and treating `Math.Sin` as an uninterpreted function (which gets an interpretation as part of the model).

As this model does not hold with the actual interpretation of `Math.Sin` over floating point values, in a second phase custom arithmetic solvers are invoked. Here, there is a single relevant arithmetic variable x, which represents the heap location `a[i]`, and a single relevant arithmetic constraint $\texttt{Math.Sin}(x) > 0.0$. The relevant constraints and variables are computed by a transitive constraint dependency analysis, starting from the constraints which involve floating point operations. To avoid reasoning about indirect memory accesses by the custom arithmetic solver, each symbolic heap location, *e.g.* `a[i]` is simply treated as a variable, ignoring further dependent constraints relating to the sub-expressions of the symbolic heap location.

Default behavior for floating point constraints: By default, Pex version 0.91 employs two custom arithmetic solvers: a solver based on trying a fixed number of (pseudo) random values, and a solver based on an early implementation of the AVM method. To conduct the experiments presented later in Section 4, all custom solvers can be selectively enabled and disabled.

3.2 Custom Solvers

The extension for the custom arithmetic solvers can be enabled by including the `PexCustomArithmeticSolver` attribute at a class, or, at the assembly level (via `[assembly: PexCustomArithmeticSolver]`) of the assembly being tested with Pex.

Fitness Function: The fitness function used by the custom solvers is defined in terms of the comparison operators that appear in the constraints, and is similar to what is commonly referred to as the *branch distance* measure in SBST [19]. The inputs to the fitness function are the original set of constraints, alongside

the current model constructed by the custom solvers. The function computes a distance value for each constraint, and the overall fitness is the sum of these values. The computation of the distance value depends on the comparison operators in the constraints, as shown in Table 1.

Table 1. Relational operators used in Pex and corresponding distance functions. K is a failure constant (set to 1).

Operator	Pex representation	Distance function
$a = b$	$a = b$	if $\|a - b\| = 0$ then 0 else $\|a - b\|$
$a \neq b$	$(a = b) = 0$	if $\|a - b\| \neq 0$ then 0 else K
$a < b$	$a < b$	if $a - b < 0$ then 0 else $a - b$
$a \leq b$	$(b < a) = 0$	if $a - b \leq 0$ then 0 else $a - b$
$a > b$	$b < a$	if $b - a < 0$ then 0 else $b - a$
$a \geq b$	$(a < b) = 0$	if $b - a \leq 0$ then 0 else $b - a$

AVM: The AVM starts by building a vector of symbolic variables that forms the basis of the exploratory moves. An index variable is used to keep track of the element in the vector currently being optimized. Further, each element has an associated *precision* variable. The precision is used to control the size of a neighborhood move, as described in Section 2.3, and is initialized to `0` for all variables, regardless of type (*i.e.* `float`,`double`, etc.).

Neighborhood Moves:
Every delta is computed as a double precision variable before being converted to the type of the variable (*i.e.* decimal, integer etc.) currently being modified. For an exploratory move, *dir* (from Section 2.3) starts off as -1. If the move is rejected, *dir* is changed to 1. If neither exploratory move is accepted (*i.e.* produces a lower fitness value than the current best model), the AVM moves on to the next element in its vector and resets *dir* to -1. Conversely, if a move is accepted, the AVM continues with pattern moves for as long as there is an improvement in fitness values. Every pattern move simply increases the variable *it* from Section 2.3 by 1. When a pattern move is rejected, the AVM restarts with exploratory moves for the first element in its vector.

Once the AVM reaches the last element in its vector, and each exploratory move of that element is rejected, the search is stuck (either on a plateau or local optima).

Precision of Floating Point Variables:
One possible cause for the search to get stuck is that the size of the neighborhood move is too coarse. Therefore, before performing a random restart, the AVM checks if the precision of a variable (see Section 2.3) can be increased. In C#, single–precision variables (*i.e.* `float`) contain about 7 decimal digits of precision, while double–precision variables (*i.e.* `double`) contains about 15 decimal digits of precision. The AVM only changes the precision ($prec_i$) for a variable, if adding

the value 10^{-prec_i} has an effect, or, $prec_i$ is less than 7 or 15 for single and double precision type inputs respectively.

Random Restarts:
The main strategy for overcoming local optima is to perform a random restart. Two types of restarts are considered. The first is a global, and the second a local restart. In a global restart, all variables in the AVM's vector are assigned new random values. This is likely to place the starting point for the next hill climb far away from the local optimum where the search got stuck. While this is desirable in many cases, it is not an ideal strategy when a global optimum is surrounded by many local optima. Chances are the search will just get stuck at the same local optimum again. Therefore, the AVM also uses a local restart, which is designed to stay in the vicinity of the current search space while still being able to escape from the optimum.

In a local restart, a random number r (between 0 and 1) is created for each variable. This number is 'scaled' by the formula $10^{-prec_i} * r$, where $prec_i$ is the current precision of the ith variable. The 'scaled' random number is then added to the existing value of the variable. If a local restart does not enable the search to make further progress, a global restart is performed. Thus, the AVM alternates between local and global restarts.

ES: Most EAs allow a wide range of configuration options and ES are no different. A user can configure the solver through various environment variables. `es_solver_parents` controls the size of the parent population μ (default: 15); `es_solver_offspring` sets the size of the offspring population (default: 100); `es_solver_recomb` sets the recombination strategy; `es_solver_mut` sets the mutation strategy.

The ES solver starts by creating an initial parent population of μ individuals. Each individual contains a model and a strategy parameter vector such that every variable in the model has a corresponding strategy parameter. The parameters are initialized to 1 and all the variables in the models are assigned random values.

The main loop of the ES solver involves recombination, mutation and selection steps. If a user specified a recombination strategy, two parents are repeatedly recombined to produce a single offspring until the offspring population is full. Then, a mutation operator is applied to each offspring in turn. The mutation operators first mutate the strategy parameter(s), and then the input variables. Once all offspring have been evaluated, they are combined with the parent population and ranked according to their fitness value. The top ranked individuals are then chosen to replace the parent population, thus forming the next generation of parents. We will now describe the reproduction operators in more detail.

Recombination: One of the genetic operators in an ES is the recombination operator where two or more parents are combined to produce one offspring. The ES solver supports the following recombination strategies:

Discrete:
In a discrete recombination, two parent individuals are chosen uniformly from

the parent population. Then, each variable in the offspring is assigned the value from either parent. Parents have an equal probability of contributing towards the offspring. The same applies for the strategy parameter vector, that is, the offspring receives each strategy parameter from either parent with equal probability.

Global Discrete:
This recombination strategy is similar to discrete recombination. The only difference is that for each variable (and each strategy parameter), a new set of parents is chosen (uniformly).

Intermediate:
During an intermediate recombination, two parents are chosen uniformly. For each variable in the offspring, the values of the corresponding parent variables are added together, and the sum is multiplied by 0.5, before being assigned to the offspring. The same is done for each strategy parameter.

Global Intermediate:
Similarly, in a global intermediate recombination, each variable and each strategy parameter are chosen from a new set of parents, before combining them in the same way as described in the intermediate strategy.

Mutation: Traditionally mutation was the main means of reproduction in ES. The custom solver supports the following mutation strategies:

Single:
In a single mutation strategy an individual only uses one strategy parameter, σ, for all variables. In addition, the mutation operator contains a learning parameter τ. It implements the following equations:

$$\tau := 1/\sqrt{n} \tag{1}$$

$$\sigma' := \sigma * \exp\,(\tau * N(0,1)) \tag{2}$$

$$v_i := v_i + \sigma' * N_i(0,1), \qquad i = 1, \ldots, n \tag{3}$$

First, the strategy parameter σ of an offspring is mutated to produce σ'. $N(0,1)$ denotes a random number from a standard univariate normal distribution. The mutated strategy parameter is then used to control the mutation of the variables. For each variable (from $1, \ldots, n$), a new random number from a standard univariate normal distribution is multiplied by σ', and then added to the existing value of the variable.

Multi:
This mutation strategy contains two learning parameters, a global one, τ_0, and a local one, τ. Further, each variable uses its own strategy parameter. The following equations are implemented:

$$\tau_0 := 1/\sqrt{2 * n} \tag{4}$$

$$\tau := 1/\sqrt{2 * \sqrt{n}} \tag{5}$$

$$\sigma_i' := \sigma_i * \exp(\tau_0 * N(0,1) + \tau * N_i(0,1)), \qquad i = 1, \ldots, n \tag{6}$$

$$v_i := v_i + \sigma_i' * N_i(0,1), \qquad i = 1, \ldots, n \tag{7}$$

4 Empirical Study

The empirical study was split into two parts. The first part contains a set of benchmark functions which are commonly used to evaluate optimization algorithms [12]. The second part consists of two real world programs comprising 152,372 lines of C# code (as reported by the SLOCCount tool [20]).

4.1 Benchmark Functions

Details of the benchmark subjects are recorded in Table 2. The optimization problem is to find the minimum of each function (*i.e.* 0). The functions were formulated as a single `if`-statement, with the `then`-branch declared as a test data generation goal for Pex. 100% block coverage of the methods indicates that the Pex goal has been reached.

Table 2. Summary of the benchmark functions and their C# representation.

Function	C# representation
Beale	(1.5 - x1 * (1 - x2)) == 0
Freudenstein And Roth	(-13 + x1 + ((5 - x2) * x2 - 2) * x2) + (-29 + x1 + ((x2 + 1) * x2 - 14) * x2) == 0
Helical Valley Function	double theta(double x1,double x2) { if(x1 > 0) return Math.Atan(x2 / x1) / (2 * Math.PI); else if (x1 < 0) return (Math.Atan(x2 / x1) / (2 * Math.PI) + 0.5); else return 0; } (10 * (x3 - 10 * theta(x1, x2))) == 0 && (10 * (Math.Sqrt(x1 * x1 + x2 * x2) - 1)) == 0 && x3 == 0
Powell	(Math.Pow(10, 4) * x1 * x2 - 1) == 0 && (Math.Pow(Math.E, -x1) + Math.Pow(Math.E, -x2) - 1.0001) == 0
Rosenbrock	Math.Pow((1 - x1), 2) + 100 * (Math.Pow((x2 - x1 * x1), 2)) == 0
Wood	(10 * (x2 - x1 * x1)) == 0 && (1 - x1) == 0 && (Math.Sqrt(90) * (x4 - x3 * x3)) == 0 && (1 - x3) == 0 && (Math.Sqrt(10) * (x2 + x4 - 2)) == 0 && (Math.Pow(10, -0.5) * (x2 - x4)) == 0

4.2 Real World Open Source Programs

Details for the two open source programs are shown in Table 3. Both programs are written in C# and were chosen specifically because they contain arithmetic operations over floating point variables. Alglib [1] is a numerical analysis and data processing library, and QLNet [13] is a library for quantitative finance operations.

Table 3. Details of the open source programs.

Program	SLOC	Pex Methods	Tested
Alglib	62,271	608	514
QLNet	90,101	3,123	3,068
Total	152,372	3,731	3,582

4.3 Experimental setup

The benchmark functions, Alglib, and QLNet source files were assembled into a single Visual Studio (VS) project each. We then created three test projects via the *Pex* $\rightarrow$ *Create Parameterized Unit Tests* command. This generates a `PexMethod` for every public method in the original source code. A `PexMethod` represents a parametrized unit test and serves as an entry point for the DSE.

If the arguments to a test function contain complex type objects, Pex may need help in constructing meaningful inputs in order to achieve a higher level of code coverage. Pex provides various mechanisms to that effect such as factory methods and the Moles framework. For the experiments, none of the generated parameterized unit tests were modified, and no factory methods or 'moled' objects were used.

The fitness budget for the random search used by Pex, the AVM and ES was limited to $100,000$ fitness evaluations. The ES was configured to have 15 parents, produce 100 offspring per generation, use a *Global Discrete* recombination strategy and the *Single* mutation operator described in Section 3.2.

Setup for Benchmarks:
Pex contains a number of options to bound its exploration. For example, the default time out for Pex's constraint solver is 1 second. We increased this limit to $10,000$ seconds for the constraint solvers (`/mcst` option) and also increased the time limit for an exploration to the same amount. Then, we ran Pex for each `PexMethod` with: 1) only the constraint solver Z3 enabled, 2) a random search and Z3 enabled, 3) the AVM and Z3 enabled, 4) the ES and Z3 enabled.

Apart from the configuration with only Z3 enabled, we repeated the runs 30 times for each benchmark, due to the stochastic nature of both the AVM and the ES. A list of 30 random number seeds was used to seed the random number generator in Pex (by setting the environment variable `er_random_seed`). Repeating experiments for a stochastic algorithm samples its behavior for a given problem, and thus reduces the risk that any observed change in effectiveness is due to chance.

Setup for Open Source Programs:
For the open source programs it was necessary to restrict the execution time of Pex and its constraint solvers to make the study scalable. Many of the functions tested contained unbounded loops and complex constraints over arithmetic types, which slow down the exploration of a program. Therefore, we allowed Pex 60 seconds per `PexMethod` (`/to` option), and set the time out for its constraint solvers to 20 seconds (`/mcst` option).

Further, because the programs comprised 3,731 functions, we did not repeat the experiments 30 times for each function. We believe that the risk of any observed difference when using the custom solvers being due to chance, is sufficiently mitigated by using such a large pool of functions.

4.4 Analysis

Benchmarks:
Figure 2 summarizes the results in terms of block coverage for the benchmark functions. Overall, the ES solver is the most effective, achieving 100% block coverage for 5 out of 6 functions. It is also the only algorithm that finds the optimum of the Rosenbrock function, which has a large, almost flat valley near the optimum. The AVM fails to reach the Pex goal for the Rosenbrock function, and also fails in 3 out of 30 runs for the Beale function. Neither solver achieves full coverage of the Powell function.

These figures show that the custom solvers can improve the coverage of Pex for floating point computations. However, they also show that the mapping between rational numbers and floating point numbers is not always a problem in practice. For 3 out of 6 functions Pex was able to achieve 100% block coverage without using any heuristics on top of Z3.

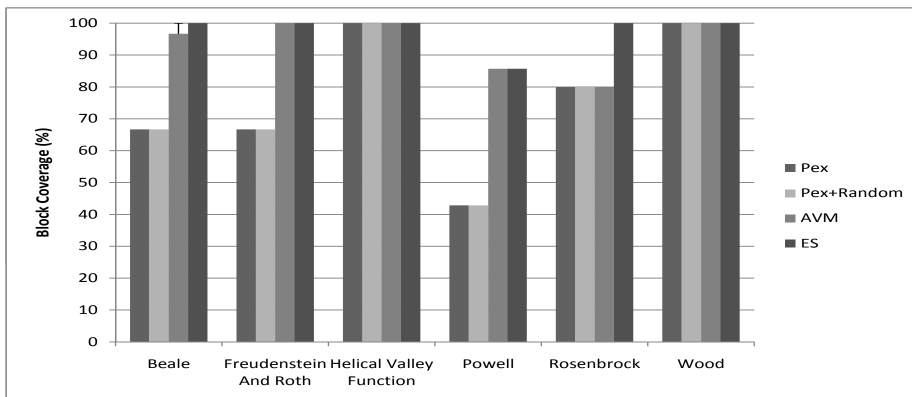

Fig. 2. Block coverage achieved with the different configurations of Pex on the benchmark problems. 100% coverage means the testing goal has been reached.

Open Source Programs:
Next, we looked at two open source programs to gain a better understanding of what benefits the custom solvers might offer to a user in practice. During this study, the random search in Pex remained enabled, and we turned on the AVM and ES solvers on top of it; *i.e.* they only get invoked when Z3 or the random search fail to find a solution.

Table 4 reports the coverage for Pex (with a random search), Pex and the AVM, and Pex and the ES solver enabled. Coverage was measured using the

reports produced by Pex and shows the ratio of blocks covered to the total number of blocks discovered by Pex (during its DSE). The 'Failed Methods' column indicates the number of methods for which Pex was terminated abnormally after 60 seconds. Those methods do not contribute to the coverage reported, *i.e.* blocks covered during runs which were terminated, are ignored.

Strikingly, the results show that the coverage with the AVM solver enabled, is less than with the custom solvers disabled. The ES solver only marginally increases the coverage compared to Pex's default solvers (*i.e.* Z3 and random search). When the custom solvers are enabled, they will, on average, take around 5 seconds to either find a solution or exhaust their fitness budget. However, as can be seen from Table 5, there are instances where the AVM (438 times) and ES (310 times) were terminated after 20 seconds. Obviously, any time spent in one of the custom solvers is no longer available to Pex to explore different parts of the program. A random search is very fast, and if it fails, Pex can quickly 'move on'. Thus, it can spend more time exploring execution paths which do not depend on floating point computations, thereby increasing the overall level of coverage.

The results from Table 4 suggest that the custom solvers proposed in this paper should only be enabled if the code is known to produce many constraints over floating point variables. In such an event, sufficient resources, in terms of fitness evaluations as well as runtime, should be allocated for the solvers to be effective (*e.g.* as was the case for the benchmark functions). One has to also bear in mind that we do not know how many of the constraints passed to the custom solvers were infact infeasible. Since the solvers have no way of checking, infeasible constraints will cause the solvers to exhaust either their fitness budget or their time out, thus wasting valuable exploration time.

Table 4. Block coverage achieved by Pex, Pex+AVM, and Pex+ES for the open source programs. Coverage is reported as a percentage of blocks discovered by Pex and blocks covered. The 'Failed Methods' columns indicate the number of methods for which Pex was terminated after 1 minute. Any blocks covered in those methods are not counted. The 'Time' column reports the total wall clock time spent inside the AVM and ES solvers.

Program	**Pex**		**Pex+AVM**			**Pex+ES**		
	Cov.	Failed Methods	Cov.	Failed Methods	Time	Cov.	Failed Methods	Time
Alglib	43.86%	32	41.04%	40	02:42:42	44.13%	77	01:53:15
QLNet	43.54%	24	44.46%	31	01:06:02	46.84%	45	00:54:39

Another interesting question is how often constraints over floating point variables are a problem for DSE in practice. Table 5 shows that, for Alglib, over a third of its methods contain constraints over floating point numbers that could not be solved by either Z3 or a random search. For the QLNet library on the other hand, this figure is much smaller at around 3% of its methods. This suggests that it very much depends on the type of application being tested. Nevertheless, the number of constraints per application (first column in Table 5) which cannot be

Table 5. This table shows how often either the AVM or ES solvers have been invoked by Pex. 'UM' stands for 'Unique Methods' and shows the number of methods that contained one or more constraints which could not be solved by Z3 or a random search. The 'Succ.' columns show how often the AVM or ES solvers were successful, while 'Timeouts' counts the number of times the AVM or ES were terminated after 20 seconds. The 'Fail' column lists the number of unsuccessful attempts by the AVM or ES to find a solution. The numbers in brackets show the number of methods for which the custom solvers were invoked, but did not receive any variables to optimize from Pex.

Program	**Pex+AVM**				**Pex+ES**			
	Inv. (UM)	Succ.	Timeouts	Fail	Inv. (UM)	Succ.	Timeouts	Fail
Alglib	1828 (220)	646	398	784 (269)	1170 (172)	521	153	496 (269)
QLNet	557 (93)	37	40	480 (0)	495 (87)	10	0	485 (0)
Success rate	28.64%				31.89%			

solved by a random search or Z3, is large enough to present a problem in practice, especially, since this number will be higher if Pex is given more exploration time.

The final part of the study was to analyze the data types of the variables involved in floating point constraints, and the arithmetic operators over them. Figure 3 shows that constraints passed to the AVM and ES solvers only included variables of type `double` and integral types (*e.g.* `short`, `int` etc.). Thus, it is important that any approach dealing with constraints over floating point variables can handle mixed floating point and integral type constraints.

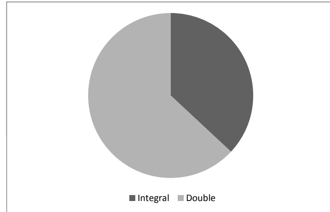

Fig. 3. Distribution of the data types of the variables passed to the AVM and ES solvers.

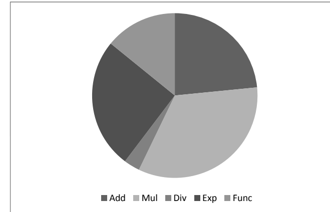

Fig. 4. Distribution of arithmetic operators in the constraints passed to the AVM and ES solvers.

Figure 4 summarizes the observed arithmetic operators in the constraints attempted by the AVM and ES solvers. Most commonly the constraints involved multiplication of input variables, followed by the power operator (*e.g.* x^y). These two operators are likely to produce non–linear constraints, and thus are especially hard for constraint solvers. The third most frequent operator was addition of two variables, followed by system library calls involving floating point variables, such as `Double.IsNaN`.

4.5 Threats to Validity

Naturally there are threats to validity in any empirical study such as this. The first issue to address is the internal validity of the experiments, *i.e.* whether there has been a bias in the experimental design that could affect the obtained results. One potential source of bias comes from the settings used for Pex and the custom solvers. For the open source programs, the time out for Pex explorations was set at 60 seconds, while constraint solvers were given 20 seconds per invocation. These limits, together with the fitness budget of 100,000 evaluations, were necessary to control the scope of the study. However, as a result, the benefit of using the custom arithmetic solvers was diminished, because the functions tested were so complex that they required longer execution times and bigger fitness budgets.

Another potential source of bias comes from the inherent stochastic behavior of the meta–heuristic search algorithms used in the custom solvers. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests using a sufficiently large sample of result data. In order to ensure a large sample size, experiments for the benchmark functions were repeated 30 times. The open source programs comprised 3,731 functions, thus also providing a large pool of data from which to draw observations.

A further source of bias includes the selection of the functions used in the empirical study, which could potentially affect its external validity, *i.e.*, the extent to which it is possible to generalize from the results obtained. The benchmark functions have been used to evaluate different optimization algorithms before, thus were considered a good candidate to test Pex and each custom solver (implementing an optimization algorithm) in the extreme. The open source programs were chosen because they represent non-trivial sized programs and because they contain complex floating point arithmetic. As with any empirical study such as this, caution is required before making any claims as to whether these results would be observed on other functions. More experiments are needed to validate or refute such claims.

5 Conclusions

This paper has presented an extension to DSE for floating point computations, based on SBST techniques. The extension has been implemented as a plug in for the Microsoft Pex DSE testing tool. Results from a set of benchmark functions show that it is possible to increase the effectiveness of what might be called "vanilla DSE". However, a study on two open source programs also shows that for the solvers to be effective, they need to be given adequate resources in terms of wall clock execution time, as well as a large fitness budget. Otherwise any increase in coverage is, at best, marginal. More experiments are needed to check if longer exploration times for Pex make the custom solvers more effective.

Acknowledgments

Kiran Lakhotia is funded by EPSRC grant EP/G060525/1. Mark Harman is supported by EPSRC Grants EP/G060525/1, EP/D050863, GR/S93684 & GR/T22872 and also by the kind support of Daimler Berlin, BMS and Vizuri Ltd., London.

References

1. Alglib. Alglib. http://www.alglib.net/.
2. B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test, Verif. Reliab*, 16(2):97–121, 2006.
3. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
4. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming language design and implementation*, pages 234–245, 2002.
5. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.
6. M. Harman, K. Lakhotia, and P. McMinn. A multi-objective approach to search-based test data generation. In *GECCO 2007*, pages 1098 – 1105, 2007.
7. M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36(2), 2010. To Appear.
8. K. Inkumsah and T. Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE 2007*, pages 425–428, 2007.
9. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
10. R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE 2007*, pages 416–426. IEEE Computer Society, 2007.
11. W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.
12. J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Testing unconstrained optimization software. *ACM Trans. Math. Software*, 7(1):17–41, 1981.
13. QLNet. QLNet. http://www.qlnet.org/.
14. I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973.
15. H.-P. Schwefel. *Numerical optimization of Computer models*. John Wiley & Sons, Ltd., Chichester, 1981.
16. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006*, pages 419–423, 2006.
17. N. Tillmann and J. de Halleux. Pex-white box test generation for.NET. In *TAP 2008*, pages 134–153, 2008.
18. P. Tonella. Evolutionary testing of classes. In *ISSTA 2004*, pages 119–128, 2004.
19. J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.
20. D. A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size. http://www.dwheeler.com/sloc/, 2001.
21. T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *International Conference on Dependable Systems and Networks (DSN 2009)*, 2009.