ELECTRONICS AND ELECTRICAL ENGINEERING
ISSN 1392 – 1215 2012. No. 6(122)
ELEKTRONIKA IR ELEKTROTECHNIKA

SYSTEM ENGINEERING, COMPUTER TECHNOLOGY
T 120
SISTEMŲ INŽINERIJA, KOMPIUTERINĖS TECHNOLOGIJOS

Generating Unit Tests for Floating Point Embedded Software using Compositional Dynamic Symbolic Execution

J. Prelgauskas, E. Bareisa
Software Engineering Department, Kaunas University of Technology,
Studentų str. 50-406, Kaunas, Lithuania, e-mails: justinas.prelgauskas@ktu.lt, eduardas.bareisa@ktu.lt

crossref http://dx.doi.org/10.5755/j01.eee.122.6.1814

Introduction

Compositional dynamic symbolic execution [1, 2] is a well-known technique for generating unit test data that would achieve high branch coverage for software with no floating point data types. However, if source code heavily depends on floating point computations, this method is unable to achieve good code coverage results. In this paper we present a method that would integrate both: compositional dynamic symbolic execution [3] and search-based test data generation methods to achieve better code coverage for embedded software that largely depends on floating point computations. We have implemented our method as an extension of a well-know symbolic execution engine – PEX [4]. Our extension implements search-based testing as an optimization technique using AVM [5] method. We present coverage comparison for several benchmark functions.

Dynamic symbolic execution

Static symbolic execution was first proposed by J. C. King [6]. However, in real-world scenarios this method suffers because of external calls, unsolvable constraints and functions that cannot be reasoned about symbolically. This is why dynamic symbolic execution (DSE) was proposed. DSE combines concrete and symbolic execution by providing constraint solver runtime values. Constraints in DSE are simplified in this way so that they become more amenable to constraint solving. However, if source code is analyzed as a single "flat" unit, this method quickly faces the state space explosion problem because its' complexity is exponential in terms of function count in analyzed program.

Compositional dynamic symbolic execution

Compositional dynamic symbolic execution is an extension to DSE and is currently implemented in several popular tools, such as PEX [4], EXE [7], SMART [1], CUTE [8]. The main idea behind compositional dynamic symbolic execution is to extend DSE with inter- and intra-function call support. Calls to internal functions are analyzed, and every function is augmented with additional meta-data – *function summaries*. Authors of SMART [1] define *function summary* φ_f for a function *f* as a formula of propositional logic whose propositions are constraints expressed in some theory *T*. φ_f can be computed by successive iterations and defined as a disjunction of formulas φ_w of the form $\varphi_w = pre_w \wedge post_w$, where pre_w is a conjunction of constraints on the inputs of *f* while $post_w$ is a conjunction of constraints on the outputs of *f*. φ_w can be computed from the path constraint corresponding to the execution path *w*. If we analyze functions in this way we will see that the number of execution paths considered by compositional dynamic symbolic execution will be at most *nb* (where *n* is the number of functions *f* in program *P, b* is the search depth bound) and is therefore linear in *nb*. However, this method would end performing random search for floating point constraints that are present in program.

Problem formulation

In order to reach a branch with some concrete input values, a path constraint for this branch must be passed to constraint solver. Then, a constraint solver must find a counter-example for this constraint or end search with answer "unsatisfiable". Currently search for a counter-example for floating-point path constraints are implemented in one of the following ways: 1) approximating floating point data types as real types and solving constraint in theory of real numbers [9] (authors' note: no SMT theory of floating point numbers was available at the moment of writing this publication) or 2) performing random search. However, both solutions have their drawbacks. First option does not take into account specifics of floating point arithmetic (rounding

error, normalized forms, etc.) [10]. This method can even sometimes return **incorrect** results. For example when approximating path constraint

$$PC(x) = \{x > 0.0, x + 1.0e^{12} = 1.0e^{12}\}, \quad (1)$$

as real constraint (despite x is a floating point variable), constraint solver would not be able to find any solutions that would satisfy this path constraint. However, any floating-point number (represented as defined in IEEE 754 standard [11]) in interval

$$[1.401298464324817e^{-45} \ldots 32767.9990234] \quad (2)$$

is a solution to this constraint. Second option (random search) does not have correctness problem, bus is not as **effective** as some search-based heuristic methods. Furthermore, random search does not have reasonable search termination criteria (it depends only on given time/memory bound of the search). This is why we formulate this problem as an optimization problem and define proper search termination criteria for this type of search (similarly as what was done for hardware testing [12]).

Optimization problem: reaching branches with unsolvable and floating-point path constraints

As soon as our method gets response from constraint solver with answer "unsatisfiable" and constraint contains floating point data types, the corresponding path constraint is forwarded to search-based AVM sub-routine. Entire path constraint (*PC)* can be formulated as a disjunction of individual branch statement constraints in the following way

$$PC(\vec{x}) = \wedge pc_i, i \in 0..N. \quad (3)$$

Vector $\vec{x}$ is what we are trying to find: concrete input values for method under test (MUT). First, we initialize this vector with all random values. Second, using AVM method, we try to move candidate solution towards optimum solution. To do this we first must define what the optimum is by defining the objective function. This function will be used to define whether new generated test input is better or not.

We define the objective (also known as fitness) function Φ as a weighted sum of *branch distance* [13] measures (δ) for each branch statement

$$\Phi(\vec{x}) = \sum_{i \in 0..N} w_i * \delta(\vec{x}). \quad (4)$$

Algorithm goal is to minimize function Φ. When our algorithm starts, all weights w_i for each branch statement pc_i are initialized to 1. The weight is increased if no solution is found for specified number of iterations. This helps algorithm to concentrate on difficult tasks, because whole solution is found only if it satisfies all clauses pc_i.

Distance function δ measures how far current candidate solution $\vec{x}$ is from satisfying the branch constraint pc_i. .NET languages may have various constraint operators in branches. We have defined distance measure function for each of these operators in Table 1.

Table 1. Distance measures for .NET branch operators

Op-code	Examples	δ	Description
BEQ	$a = b$	$N(\lvert a - b \rvert)$	Branch on equal.
BGE	$a \geq b$	IF $a \geq b$ THEN 0 ELSE $N(b - a)$	Branch on greater than or equal to.
BGT	$a > b$	IF $a > b$ THEN 0 ELSE $N(b - a)$	Branch on greater than.
BLE	$a \leq b$	IF $a \leq b$ THEN 0 ELSE $N(a - b)$	Branch on less than or equal to.
BLT	$a < b$	IF $a < b$ THEN 0 ELSE $N(a - b)$	Branch on less than.
BNE	$a \neq b$	IF $a \neq b$ THEN 0 ELSE 1	Branch on not equal.
BRFALSE	$a = null$ $a = FALSE$ $a \leq 0$	IF $a = null \mid$ $a = FALSE \mid$ $a = 0$ THEN 0 ELSE 1	Branch on false, null, or zero.
BRTRUE	$a \neq null$ $a = TRUE$ $a > 0$	IF $a = null \mid$ $a = FALSE \mid$ $a = 0$ THEN 1 ELSE 0	Branch on non-false.

Function *N* denotes a denormalisation function, which normalizes its input into interval [0..1].

Evaluation

We have implemented proposed method as an extension to a well known dynamic symbolic execution tool PEX [4]. Extension was implemented as a custom arithmetic solver using interface *Microsoft.ExtendedReflection.Reasoning. ArithmeticSolving.IArithmeticSolver.* Our extension can be enabled by simply adding custom *AVMCustomArithmeticSolver* attribute to the test assembly containing parameterized unit tests.

Evaluation subjects – benchmark functions

We have evaluated our method with several classical optimization problems [14], manually rewritten into C#

language, .NET Micro framework. To implement benchmark functions we needed several mathematical functions (such as Pow, Atan, Sqrt, Exp) that are not implemented in .NET Micro framework. For this purpose we will use our extended version of Microsoft.SPOT.Math class – "exMath".

Table 2. Benchmark functions

Benchmark	C# representation
Brown badly scaled function	(x1 - exMath.Pow(10, 6)) == 0 && (x2 - 2 * exMath.Pow(10, -6)) == 0 && (x1 * x2 - 2) == 0
Beale function	(1.5 - x1 * (1 - x2)) == 0
Powell singular function	((x1 * 10 * x2) == 0) && (exMath.Pow(5, 0.5) * (x3 - x4) == 0) && (exMath.Pow((x2 - 2 * x3), 2) == 0) && (exMath.Pow(10, 0.5) * exMath.Pow((x1 - x4), 2) == 0)
Freudenstein and Roth function	((-13 + x1 + ((5 - x2) * x2 - 2) * x2) == 0) && ((-29 + x1 + ((x2 + 1) * x2 - 14) * x2) == 0)
Rosenbrock function	((1 - x1)==0)&&(10*(x2 - x1 * x1) == 0)
Helical Valley function	double theta(double x1,double x2) f if(x1 > 0) return exMath.Atan(x2 / x1) / (2 * exMath.PI); else if (x1 < 0) return (exMath.Atan(x2 / x1) / (2 * exMath.PI) + 0.5); else return 0; g (10 * (x3 - 10 * theta(x1, x2))) == 0 && (10 * (exMath.Sqrt(x1 * x1 + x2 * x2) - 1)) == 0 && x3 == 0
Powell badly scaled function	((exMath.Pow(10, 4) * x1 * x2 - 1) == 0) && ((exMath.Exp(-x1) + exMath.Exp(-x2) - 1.0001) == 0)
Wood function	(10 * (x2 - x1 * x1)) == 0 && (1 - x1) == 0 && (exMath.Sqrt(90) * (x4 - x3 * x3)) == 0 && (1 - x3) == 0 && (exMath.Sqrt(10) * (x2 + x4 - 2)) == 0 && (exMath.Pow(10, -0.5) * (x2 - x4)) == 0

Experimental setup

We have implemented simple parameterized test for each benchmark function. Parameterized tests would take function inputs as arguments and have only one "IF" statement. If global optimum for given benchmark function is reached, then test would pass, otherwise, Assert.Fail() will be called. Because of stochastic nature of experiment, we repeated each iteration 100 times. In every iteration for every benchmark function we performed following actions: generated available test inputs, run the generated test and measured block coverage. Average block coverage for each function is given in Fig. 1. In order to have proper results we had to limit exploration bounds. We set maximum exploration time bound for both random and AVM methods to 1 minute.

Coverage increase

We can clearly see from Fig. 1, that our proposed method outperforms random search in terms of branch coverage for most of benchmark functions. The only function for which average branch coverage was not increased using AVM method was Rosenbrock. Function itself has a long, parabolic shaped flat valley. The global minimum is inside that valley. AVM method finds valley easily, however to converge to the global minimum given previously defined exploration bounds is difficult.

For Beale and Powell badly scaled functions our proposed method was able to find solution, but not for all iterations. This is why average block coverage for those functions was not 100%.

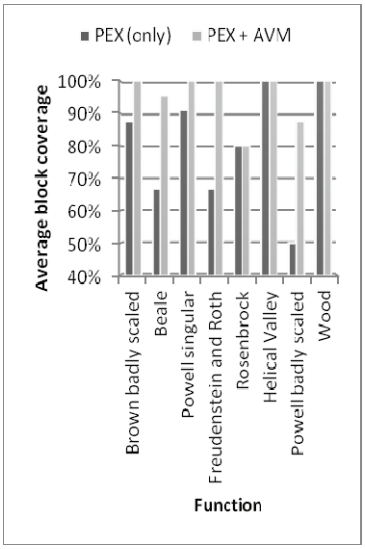

Fig. 1. Average block coverage of generated tests

Conclusions

In this article we presented a method for generating test for embedded software that is highly dependent on floating-point expressions and calculations. We have combined two previously well known techniques to solve this problem: compositional dynamic symbolic execution for path exploration and search-based testing (using alternating variable method) for complex floating-point computations.

We have evaluated our proposed method against a suite of benchmark functions, compiled in .NET Micro Framework. In our future work we plan to implement more algorithms as custom arithmetic solvers, compare their performance and coverage increase. Furthermore, there has been several attempts to integrate floating point arithmetic into several well know SMT solvers (such as Z3, JPF). We plan to evaluate same test subjects as soon as any of these SMT solvers gets updated with floating-point solving capabilities. There has recently been issued several studies on real-world software [15, 16] and studies showed, that approximating floating-point types as real types is not always an issue. This is why we plan to perform further experiments with real-world open source projects as well as other optimization algorithms [17].

References

1. **Godefroid P.** Compositional dynamic test generation // Proceedings of the 34th annual ACM SIGPLAN–SIGACT symposium on Principles of programming languages. – ACM, New York, USA, 2007. – P. 47–54.

2. **Anand S., Godefroid P., Tillmann N.** Demand–driven compositional symbolic execution // Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems. – Springer–Verlag, Berlin, Heidelberg, 2008. – P. 367–381.
3. **Vanoverberghe D., Piessens F.** Theoretical aspects of compositional symbolic execution // Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science. – Springer–Verlag, Berlin, Heidelberg, 2011. – No. 6603/2011. – P. 247–261.
4. **Tillmann N., De Halleux J.** Pex–white box test generation for. net // Tests and Proofs, Lecture Notes in Computer Science. – Springer–Verlag, Berlin, Heidelberg, 2008. – No. 4966/2008. – P. 134–153.
5. **Arcuri A.** Full theoretical runtime analysis of alternating variable method on the triangle classification problem // Proceedings of 1st International Symposium on Search Based Software Engineering. – IEEE, 2009 – P. 113–121.
6. **King J. C.** Symbolic execution and program testing // Communications of the ACM. – ACM, New York, USA, 1976. – No. 19(7). – P. 385–394.
7. **Cadar C., Ganesh V., Pawlowski P. M., *et al.*** EXE: automatically generating inputs of death // ACM Transactions on Information and System Security (TISSEC). – ACM, New York, USA, 2008. –No. 12(2). – P. 1–10.
8. **Sen K., Marinov D., Agha G.** CUTE: A concolic unit testing engine for C // Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. – ACM, New York, USA, 2005. – P. 263–272.
9. **Anand S., Păsăreanu C., Visser W.** JPF–SE: A Symbolic Execution Extension to Java PathFinder // Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science. – Springer–Verlag, Berlin, Heidelberg, 2007. – No. 4424/2007. – P. 134–138.
10. **Goldberg D.** What every computer scientist should know about floating–point arithmetic // ACM Computing Surveys (CSUR). – ACM, New York, USA, 1991. – No. 23(1). – P. 5–48.
11. **Kahan W.** IEEE standard 754 for binary floating–point arithmetic // Lecture Notes on the Status of IEEE. – IEEE, 1996. – No. 754.
12. **Bareiša E., Jusas V., Motiejūnas K., et al.** Defining Random Search Termination Conditions // Electronics and Electrical Engineering. – Kaunas: Technologija, 2006. – No. 2(66). – P. 26–31.
13. **Wegener J., Baresel A., Sthamer H.** Evolutionary test environment for automatic structural testing // Information and Software Technology. – Elsevier, 2001. – No. 43(14). – P. 841–854.
14. **Moré J. J., Garbow B. S., Hillstrom K. E.** Testing unconstrained optimization software // ACM Transactions on Mathematical Software (TOMS). – ACM, New York, USA, 1981. – No. 7(1). – P. 17–41.
15. **Lakhotia K., Tillmann N., Harman M., et al.** FloPSy – Search–Based Floating Point Constraint Solving for Symbolic Execution // Testing Software and Systems, Lecture Notes in Computer Science. – Springer–Verlag, Berlin, Heidelberg, 2010. – No. 6435/2010. – P. 142–157.
16. **Souza M., Borges M., d'Amorim M., *et al.*** CORAL: Solving Complex Constraints for Symbolic PathFinder // NASA Formal Methods, Lecture Notes in Computer Science. – Springer–Verlag, Berlin, Heidelberg, 2011. – No. 6617/2011. – P. 359–374.
17. **Misevičius A.** Generation of grey patterns using an improved genetic evolutionary algorithm: some new results // Information Technology And Control. – Kaunas: Technologija, 2011. – No. 40(4). – P. 330–343. DOI: 10.5755/j01.itc.40.4.983.

Received 2012 02 24
Accepted after revision 2012 04 25

J. Prelgauskas, E. Bareisa. Generating Unit Tests for Floating Point Embedded Software using Compositional Dynamic Symbolic Execution // Electronics and Electrical Engineering. – Kaunas: Technologija, 2012. – No. 6(122). – P. 19–22.

We present a method that would integrate both: compositional dynamic symbolic execution and search-based test data generation methods to achieve better code coverage for software that largely depends on floating point computations. We have implemented our method as an extension of a well-know symbolic execution engine – PEX. Our extension implements search-based testing as an optimization technique using AVM method. We present coverage comparison for several benchmark functions. Ill. 1, bibl. 17, tabl. 2 (in English; abstracts in English and Lithuanian).

J. Prelgauskas, E. Bareiša. Vienetų testų generavimas slankiojo kablelio įterptinei programinei įrangai kompozicinio dinaminio simbolinio vykdymo metodu // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2012. – Nr. 6(122). – P. 19–22.

Straipsnyje pristatomas metodas, integruojantis du jau esamus metodus (kompozicinį dinaminį simbolinį vykdymą ir paieška paremtą testų duomenų generavimą) tam, kad testų generatorius pasiektų geresnę kodo padengtį programinei įrangai, naudojančiai slankiojo kablelio skaičiavimus. Metodui patikrinti jį realizavome kaip gerai žinomo simbolinio vykdymo įrankio PEX įskiepį. Įskiepis paiešką atlieka AVM metodu. Darbe siūlomo metodo kodo padengtis palyginama su įprastu simboliniu vykdymu. Testavimo objektais pasirinkta keletas tipinių funkcijų, naudojamų optimizavimo programai testuoti. Il. 1, bibl. 17, lent. 2 (anglų kalba; santraukos anglų ir lietuvių k.).