DyTa: Dynamic Symbolic Execution Guided with Static Verification Results

Xi Ge[1] Kunal Taneja[1] Tao Xie[1] Nikolai Tillmann[2]
[1]Dept. of Computer Science, North Carolina State University, Raleigh, NC
[2]Microsoft Research, One Microsoft Way, Redmond, WA
{xge, ktaneja}@ncsu.edu, xie@csc.ncsu.edu, nikolait@microsoft.com

ABSTRACT

Software-defect detection is an increasingly important research topic in software engineering. To detect defects in a program, static verification and dynamic test generation are two important proposed techniques. However, both of these techniques face their respective issues. Static verification produces false positives, and on the other hand, dynamic test generation is often time consuming. To address the limitations of static verification and dynamic test generation, we present an automated defect-detection tool, called DyTa, that combines both static verification and dynamic test generation. DyTa consists of a static phase and a dynamic phase. The static phase detects potential defects with a static checker; the dynamic phase generates test inputs through dynamic symbolic execution to confirm these potential defects. DyTa reduces the number of false positives compared to static verification and performs more efficiently compared to dynamic test generation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Software Engineering, Verification, Reliability

Keywords

defect detection, testing, static analysis

1. INTRODUCTION

Software testing is a common technique used to detect defects in the program under analysis. Despite wide adoption, testing is labor-intensive and time-consuming. To address the preceding issue, various dynamic test generation techniques based on dynamic symbolic execution (DSE) have been proposed [3, 8]. However, due to the exponential path-exploration space of DSE, it is often an expensive task to use it to generate test inputs that achieve high code coverage or detect defects.

Unlike testing, static verification techniques [5, 2] analyze a software representation of the program under analysis such as source code or abstract model to find defects without actually executing the program. However, due to the conservative nature of static verification techniques, many potential defects detected by static verification techniques turn out to be false positives, posing barriers for applying these techniques in real-world circumstances.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
ACM 978-1-4503-0445-0/11/05.

To address these issues faced by static verification or dynamic test generation, we present a defect-detection tool, called DyTa, that combines both static verification and dynamic test generation. DyTa consists of a static phase and a dynamic phase. In the static phase, Dyta applies a static verification technique [6] to analyze the program under analysis. The output of this analysis is a warning report that describes potential defects in the program under analysis with their locations and descriptions.

In the dynamic phase, DyTa applies DSE [3], a state-of-the-art dynamic test generation technique. DSE starts program exploration with some default or random inputs. DSE then collects constraints on program inputs from the predicates at the executed branching statements in the program. These constraints at branching statements are referred to as branch conditions. The conjunction of all the branch conditions in the path followed during execution of an input is referred to as a path condition. DSE keeps track of the previous executions to build a dynamic execution tree. DSE, in the next run, chooses one of the unexplored branching points[1] in the execution tree (dynamically built thus far). DSE flips the chosen branching point to generate a new input that follows a new execution path. DSE repeats this process until full code coverage is achieved or user-specified bounds are reached.

Additionally, in the dynamic phase, DyTa guides DSE to explore the program under analysis by using information from the static phase. DyTa uses locations of the statically detected potential defects to prune irrelevant branches for flipping, so that DSE could cover these potential defects more efficiently, DyTa extracts and uses contract-violation conditions (i.e., error conditions) from the descriptions of the potential defects, so that DSE could be guided to satisfy these contract-violation conditions.

2. APPROACH

Our DyTa tool developed for the .NET Framework using C# accepts as inputs the program under analysis and a warning report produced by a static checker for the program under analysis. The output of DyTa is a set of confirmed defects from the warning report. Figure 1 shows a screenshot of DyTa. The source code for DyTa is available at `http://pexase.codeplex.com/`. A website of the DyTa project is available at `https://sites.google.com/site/asergrp/projects/dyta`.

2.1 Static Phase

In the static phase, DyTa analyzes and manipulates the program

[1]A branching point in the execution tree of a program is an instance of a conditional statement in the source code. A branching point consists of two sides (or more than two sides for a switch statement): the true branch and the false branch. Flipping a branching point is flipping the execution of the program from the true (or false) branch to the false (or true) branch. Flipping a branching point for a switch statement is flipping the execution of the current branch to another unexplored branch.

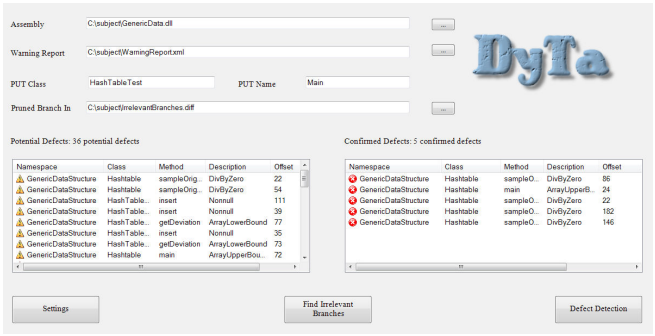

Figure 1: A screenshot of DyTa

under analysis statically. In this phase of DyTa, there are two steps: potential-defect detection and code instrumentation.

Potential-Defect Detection. The Code Contracts tool [7, 6] developed by Microsoft Research allows users to specify preconditions, postconditions, and class invariants for C# code. The static checker of Code Contracts is a static verification technique that analyzes a program, reasons abstractly about unknown variables, and tries to locate potential violations of contracts. In addition to user-defined contracts, the static checker considers a set of primitive C# operations as implicit violations of contracts such as dereferencing a null pointer and division by zero. By applying the static checker to the program, DyTa identifies a set of potential defects and collects their locations and descriptions. The descriptions include the types of potential defects and suggested preconditions to prevent them from happening (suggested preconditions are not available when inferring them is beyond the capability of Code Contracts' static checker). In the first two columns of Table 1, we show example descriptions for several potential defects. The view of listing potential defects in the left-bottom region of Figure 1 shows a summary of the warning report generated by the static checker for a hashing algorithm implementation [4]. The five columns show the namespace, class, method, description, and offset[2] of potential defects, respectively.

Code Instrumentation. DyTa applies a DSE-based test-generation tool Pex [10] in the dynamic phase (explained in more detail in the next section). In order to make the program under analysis more amenable for guided Pex exploration, DyTa instruments the program under analysis. There are two types of instrumentations that DyTa performs. (1) DyTa inserts a `PexAssume.IsTrue` method invocation at the beginning of a public method for which a precondition is suggested by the step of potential-defect detection. `PexAssume.IsTrue` specifies constraints upon test inputs generated by Pex. By adding the `PexAssume.IsTrue` invocation, DyTa filters out test inputs that are impossible to trigger the potential defects to be runtime failures. The instrumented `PexAssume.IsTrue` method invocation takes as the argument the disjunction of negations of the suggested preconditions in the preceding step. Suppose that PC_i is one of the preconditions suggested by Code Contracts in the preceding step for a public method, DyTa inserts `PexAssume.IsTrue(`$!PC_1 \| !PC_2 \| ... \| !PC_{n-1} \| !PC_n$`);` at the very beginning of this public method. (2) DyTa inserts a `BeforeWarning` method invocation right before the location of each potential defect. The `BeforeWarning` method serves as two major purposes: guiding Pex exploration with additional branching points that specify contract-violation conditions, and reporting the confirmation of the corresponding potential defects if such conditions are satisfied. The `BeforeWarning` method has three arguments: a predicate indicating the contract-violation condition, an integer specifying the contract type, and an integer specifying the instrumentation ID. DyTa attains the contract type by parsing the warning report generated by the static checker, and attains the contract-violation condition by analyzing the source code at the locations of potential defects. DyTa assigns a globally unique integer number to each `BeforeWarning` method invocation as an instrumentation ID[3]. The last two columns in Table 1 show the contract type and contract-violation condition arguments of `BeforeWarning` corresponding to each example potential defect.

[2]The offset of a given potential defect is its line number inside the intermediate language (IL) file where the potential defect lies.

Type of Potential Defect	Suggested Pre-condition	Contract Type	Contract-violation condition
Dereferencing a null pointer	object != null	Null_Pointer	object == null
Possible division by zero	Not available	Div_By_Zero	a == 0
Array access might be above the upper bound	Not available	Array_High	index >= this.values.Length
Array access might be below the lower bound	index +3 >= 0	Array_Low	index < 0

Table 1: Example potential-defect description and attained arguments for a BeforeWarning method

2.2 Dynamic Phase

In the dynamic phase, DyTa applies Pex to the program under analysis. Pex iteratively generates test inputs to cover various feasible paths. In particular, Pex flips some branching points from previous runs to generate test inputs for covering new paths. Which branching points are flipped next depends on the chosen search strategy, such as depth-first search (DFS) or breadth-first search (BFS). In our experience of applying Pex on real-world code bases, we identify that Pex cannot explore the entire program due to exponential path-exploration space. To address this issue, DyTa focuses primarily on potential defects detected by the static checker.

In particular, to confirm the potential defects reported by the static checker, DyTa often needs not flip many branching points in the program under analysis. Pruning these branching points could reduce Pex's exploration space and improve test generation efficiency. Here we adopt our previous work [9] to prune these irrelevant branches. In particular, by statically analyzing the control flow graph of the program under analysis, DyTa finds out the set of branching points whose the other unexplored branch does not lead to any location of potential defects. Such branching points are identified and would not be flipped during path exploration.

3. EXAMPLE

We next explain our approach using the code snippet shown in Figure 2. In the static phase of DyTa, the static checker reports a potential defect about array access below a lower bound at Line 10 in the `testme` method shown in Figure 2. DyTa next instruments this code snippet to facilitate the subsequent dynamic phase. In this particular case, Code Contracts does not provide any suggested precondition; therefore, only instrumentation for inserting a `BeforeWarning` method invocation is applicable. Before the location of the potential defect at Line 10, DyTa inserts a `BeforeWarning` method invocation with the arguments of the contract-violation condition, contract type, and instrumentation ID. Figure 2 shows the instrumented version of the code snippet on its left-hand side. Figure 3 also shows the method body of the `BeforeWarning` method on its right-hand side.

After the static phase, DyTa statically analyzes the control flow graph of the code under analysis to find out the branching points whose flipping could be safely pruned. There are three branching

[3]The `BeforeWarning` method uses an instrumentation ID to retrieve the information of the corresponding potential defect. DyTa maintains a table that relates each instrumentation ID with the entry in the warning report of the potential defect.

```
1:        public int testme(int a, int b)
2:        {
3:             int index;
4:             if (a > -3)
5:             {
6:                  if (b >= 10)
7:                       index = b % 10;
8:                  else
9:                       index = b;
10:                 return value[index];
11:            }
12:            return 0;
13:       }
```

Figure 2: Example code snippet under analysis before instrumentation.

```
1:int testme(int a, int b)
2:{
3:     int index;
4:     if (a > -3)
5:     {
6:          if (b >= 10)
7:               index = b % 10;
8:          else
9:               index = b;
10:           BeforeWarning
11:           (index < 0,
12:           Array_Low,
13:           31);
14:         return value[index];
15:    }
16:    return 0;
17:}

21:void BeforeWarning(bool condition,
22:           int type,
23:           int IID)
24:{
25:    if (condition)
26:    {
27:          LogConfirmedDefect(IID);
28:          switch (type)
29:          {
30:          ...
31:          case Array_Low:
32:          throw new
33:          IndexOutOfRangeException();
34:          ...
35:          }
36:    }
37:}
```

Figure 3: Example code snippet under analysis after instrumentation and the BeforeWarning method body.

points in total for path exploration (including the predicate for the contract-violation condition inside the `BeforeWarning` method), which are at Lines 4, 6, and 25, respectively. By traversing the control flow graph, DyTa finds out that when the false branch of the branching point at Line 4 is taken by a certain test execution, there would be no location of a potential defect being reached. Therefore, flipping the true branch of the branching point at Line 4 could be safely pruned.

In the dynamic phase, DyTa applies Pex to the instrumented `testme` method in Figure 3 with a pruned exploration space. In the first run, Pex arbitrarily chooses test input `a=0` and `b=0`. Therefore, the initial run takes $true$, $false$, and $false$ branches at Lines 4, 6, and 25, respectively, and the path condition collected by Pex is `a>-3 && b<10 && condition==false`.

Pex next tries to come up with a new test input by flipping a branching point. Since flipping the true branch of the branching point at Line 4 has already been pruned, Pex has to flip only either the branching point at Line 6 or Line 25. If the DFS search strategy is adopted, Pex would try to flip the branching point at Line 25, and the resulting path constraint is `a>-3 && b<10 && condition==true`. By consulting the underlying constraints solver with the new path constraint, Pex gets test input `a = 0` and `b = int.MinValue`. Executing the code snippet with such input, the `BeforeWarning` method finds out that the specified contract-violation condition is satisfied. Therefore, the `BeforeWarning` method logs that the corresponding potential defect is confirmed and throws an index out of range exception. In the worst case, if the BFS search strategy is adopted, Pex would try to flip the branching point at Line 6 first, and therefore takes one more run to confirm the potential defect at Line 14.

By using Code Contracts alone, we are unable to confirm the existence of a real defect at Line 10 in Figure 2. In contrast, DyTa reduces false positives and reports real defects with full confidence.

By using Pex alone, we have a larger and less constrained exploration space. On average, more runs are needed to confirm or detect potential defects. In contrast, DyTa provides a more constrained exploration space by considering the locations and the contract-violation conditions of potential defects.

4. RELATED WORK

Csallner and Smaragdakis proposed Check n' Crash [1], which uses a static verification tool to infer abstract error conditions at the program-input level and generates concrete test inputs from them. Check n' Crash first uses a constraint solver to generate inputs that satisfy the abstract error conditions at the program-input level, and then executes the program under analysis with such test inputs. If failures are observed, potential defects are confirmed. One drawback of Check n' Crash is that a generated test input cannot guarantee that the potential defect could be covered in the first place. To address this issue, DyTa considers both locations of potential defects and error conditions at these locations, so that test inputs could be generated with higher effectiveness.

5. CONCLUSION

Defect detection is a common process used to build high-quality software. Much attention from both industry and academia is drawn to automation of this process [3, 8, 5, 2]. Defect detection is commonly accomplished with static verification and dynamic test generation. However, both of these approaches face their respective issues. To address these issues, we present a defect-detection tool called DyTa that combines static verification and dynamic test generation. DyTa takes advantage of dynamic test generation to reduce false positives, and takes advantage of static verification to guide exploration for dynamic test generation.

Acknowledgments. This work is supported in part by NSF grants CNS-0716579, CCF-0725190, CCF-0845272, CCF-0915400, CNS-0958235, an NCSU CACC grant, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

6. REFERENCES

[1] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining Static Checking and Testing. In *Proc. ICSE*, pages 422–431, 2005.

[2] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective Typestate Verification in the Presence of Aliasing. In *Proc. ISSTA*, pages 133–144, 2006.

[3] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. PLDI*, pages 213–223, 2005.

[4] hashing algorithms, 2010. `http://hashlib.codeplex.com/`.

[5] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Proc. OOPSLA*, pages 132–136, 2004.

[6] F. Logozzo. Practical Verification for the Working Programmer with Code Contracts and Abstract Interpretation (Invited Talk). In *Proc. VMCAI*, 2011.

[7] Microsoft Code Contracts, 2010. `http://research.microsoft.com/en-us/projects/contracts/`.

[8] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. ESE/FSE*, pages 263–272, 2005.

[9] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided Path Exploration for Regression Test Generation. In *Companion Proc. ICSE, NIER*, pages 311–314, 2009.

[10] N. Tillmann and J. de Halleux. Pex: White Box Test Generation for. NET. In *Proc. TAP*, pages 134–153, 2008.