ABSTRACT

Many automatic test data generation techniques have been proposed in the past decades. Each technique can only deal with very restrictive data types so far. This limits the usefulness of test data generation in practice. We present a preliminary approach on hybrid test data generation, by combining Random Strategy (RS), Dynamic Symbolic Execution (DSE), and Search-based Strategy (SBS). It is expected to take advantage of the state-of-the-arts to enhance the robustness and scalability, in terms of different types of test data.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Verification, Reliability

Keywords
Hybrid strategy, test data generation, random strategy, dynamic symbolic execution, search-based strategy

1. INTRODUCTION

Test data generation is an important and expensive task in software testing. Many automatic techniques have been proposed to reduce the cost of test data generation. These techniques can be roughly classified into three types: Random Strategy (RS), Dynamic Symbolic Execution (DSE), and Search-based Strategy (SBS). Unfortunately, almost all of them only work with very restrictive data types. This significantly diminishes their applicability to real-life software testing.

In this poster, we present a preliminary approach on hybrid test data generation. The hybrid strategy is an intuitive idea by combining the advantages of the state-of-the-arts. However, it is a non-trivial work on hybrid test data generation, because of the huge difference and heterogeneity of RS, DSE and SBS. In our approach, we first generate test data randomly as seeds, and then generate new test data with the help of DSE and SBS on different data types. Some novel treatments are supplemented to our approach to make it robust and scalable.

2. APPROACH

In order to motivate our approach, we summarize the techniques used in our hybrid approach in Table 1, which shows the advantages and disadvantages of RS, DSE and SBS (since DSE gives more accurate integer values, we put int only in DSE to be more evident). Random testing is a cost-effective method which can randomly generate massive data, but the generated data will be unstable and redundant. Most of the existing symbolic execution techniques [8] only support integer types precisely and cost-efficiently. Although some symbolic execution techniques have tried to tame other types, mainstream constraint solvers do not work well enough on them in practice.

To be specific, we list some typical types as examples. (1) Most constraint solvers handle floating-point numbers as rational numbers, namely using a different storage format, as a result we may miss the bugs or cannot get high coverage. Constraint solver with capability of handling floating-point numbers was proposed, though, it has the limitation when facing expressions with a combination of integer and floating-point numbers [2]. (2) Constraint solvers handle strings using array-like representation (bitvector) or automata. Using bitvector, if strings with current length cannot satisfy the constraint solver, the length needs to be adjusted and the constraint needs re-solving, which increases the cost a lot. As for the automata approach, we suffer from the in-house weakness of automata approach when dealing with negative constraints (i.e. notEquals etc.).

While constraint solvers have their own weaknesses in dealing with different data types, for data types like float and character string types, we choose SBS to supplement the capability because it is not confined by the inadequacy of constraint solver, and have the ability to generate precise data for float and string types.

In summary, any one of these technique is far from enough for test data generation. Consequently, hybrid strategy can be an emerging direction. Though many previous works were proposed [7] [6], they often use one technique to enhance the other. What we want to propose is an efficient and usable approach absorbing the strength of each tech-
Techniques | Features | Advantages | Disadvantages | Data Types | References
---|---|---|---|---|---
RS | Random | Cost-effective | Unstable | * | [3]
DSE | Constraints of paths | Structural information | Path explosion, complex constraints | int, enums | [3] [8]
SBS | Objective oriented | Scalability | Local optima, search landscape | float, string | [1] [5]

Table 1: Overview of approach

It maintains a population of solutions, which are iteratively recombined and mutated to evolve successive populations. As for our approach, the fitness function in our approach is string ordinal distance, and the mutation operators are deletion, insertion and substitution [1]. Specifically, deletion operator is defined to delete a random character from string, insertion operator is defined to insert a randomly generated character into a random position, along with substitution operator, defined to replace a character in string with another with similar ordinal value.

For others (i.e., $pc_o$), we provide users with user-defined elaboration algorithm since many data structures are also user-defined. Default treatment will be used by SBS, offering users a “shell” to define the fitness function to judge the quality and evolution operator to get newcomers.

Test data generation: After we get the result obtained in the previous step, we substitute variables in the original $pc$ with them, which makes the original $pc$ easier to be solved. Follow on, the updated $pc$ will solved by constraint solvers. The generated data is the input of the next iteration.

In summary, hybrid strategies have been successfully applied in many fields because of inadequacy of single technique. In this poster, we present an approach that combines the advantages of the state-of-the-arts. We use random testing to gain initial input swiftly, DSE to acquire inputs which can guide the execution and SBS to get higher precision and fitness on different data types. The approach provides automatic testing with better robustness and scalability.

3. REFERENCES

Figure 1: Framework

Guided Negation: The path constraints ($pc$) split the input domain into separate partitions. To force the program through a different partition, a new $pc$ is obtained by negating the last predicate of the current $pc$ [3]. If the new $pc$ is solvable, we backtrack to the previous predicate. Not until the process has gotten stuck or all paths has been traversed can we reset the process to the first step.

Extraction of $pc$: The generated $pc$ is resolved in terms of typical data types. Parts consist of floating-point numbers and strings are extracted from the $pc$ respectively. Suppose $pc = pcf \land pcs \land pc_o$, in which $pcf$ is the part of $pc$ that contains float types, $pcs$ is the string constraints and $pc_o$ is other types that can be extended by users.

Elaboration: For each data type, i.e. $pcf$, $pcs$, and $pc_o$, we use proper means to handle them.

For floating-point numbers, i.e. $pcf$, we use AVM [4] to elaborate the precision of numbers, on account of the inadequate capacity of constraint solver among all floating-point numbers. AVM begins with an exploratory phase, where one of the inputs makes moves by adding or subtracting a delta. Once the moves lead to an improved fitness value computed from a defined fitness function, a pattern phase is entered. A larger number will be added or subtracted to the input variable, namely leads to an accelerated move until an optimum is reached. When no further improvements can be found for an input variable, other input parameters will be optimized in turn following similar rules.

With respect to the strings ($pc_o$), we use Genetic Algorithm (GA), a natural fit with string, to handle the problem. GA evolves a population of candidate solutions, and a fitness function which guides the search to generate test data in order to achieve a given criterion (e.g. coverage, cost).