Engineering Applications of Artificial Intelligence 45 (2015) 441–452

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai

The application of iterative interval arithmetic in path-wise test data generation

Ying Xing *, Yun-Zhan Gong, Ya-Wen Wang, Xu-Zhou Zhang

State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China

ARTICLE INFO

Article history:
Received 12 December 2014
Received in revised form
31 May 2015
Accepted 27 July 2015
Available online 27 August 2015

Keywords:
Test data generation
Constraint satisfaction problem
Interval arithmetic
Arc consistency
AC-3

ABSTRACT

Research of path-wise test data generation is crucial and challenging in the area of software testing, which can be formulated as a constraint satisfaction problem. In our previous research, a look-ahead search method has been proposed as the constraint solver for path-wise test data generation. This paper analytically studies interval arithmetic of the search method in detail, which enforces arc consistency, and introduces the iterative operator to improve it, aiming at detecting infeasible paths as well as shortening generation time. Experiments were conducted to compare the iterative operator with the classical look-ahead operator AC-3, and to compare the test data generation method using the iterative operator with some currently existing methods. Experimental results validate the effectiveness and practical deployment of the proposed iterative method, and demonstrate that it is applicable in engineering.

1. Introduction

Automating the process of software testing is a very popular research topic and is of real interest to the industry (Bertolino, 2007; Elsayed, 2012), because manual testing is time-consuming and error-prone, and is even impracticable for large-scale real-world programs in engineering (Weyuker, 1999). As a basic problem in software testing, the automation of path-wise test data generation is particularly important since many problems in software testing can be transformed into it, and people have put great efforts in this field both commercially and academically.

Currently, most of the commercial tools implement a test data generation strategy that uses constant values found in the program under test (PUT) or values that are slightly modified by means of mathematical operations (Galler and Aichernig, 2013). Take *C++ test* for example, which is a commercial software quality improvement tool for *C/C++*, it selects randomly a value from a predefined pool of values, such as minimum and maximum values, -1, $+1$ and 0 for integer types, and constant values given within the PUT. But when this kind of constraint solvers reaches their limitations, they use random-based techniques, indicating that they are not totally intelligent and automatic.

From an academic point of view, the problem of path-wise test data generation can be formulated into a constraint satisfaction problem (CSP) (Shan et al., 2004). For the purpose of solving the CSP, it is required to abstract the constraints to be met, and propagate and solve these constraints to obtain the test data. It is also strongly demanded to have precision in generating test data and the ability to prove that some paths are infeasible. DeMilli and Offutt (1991) proposed a fault-based technique using algebraic constraints and bisection to describe test data designed to find particular types of faults. Adtest (Gallagher et al., 1997) only considered one predicate or one input variable and iterated the solving procedures, which was relatively inefficient and not suitable for real-world programs in engineering. Gupta et al. (1999) presented a program execution-based approach to generate test data for a given path. The technique derived a desired input for a test path by iteratively refining an arbitrarily chosen input. Robschink and Snelting (2002) statically converted the program into a Static Single Assignment (SSA), normally resulting in large constraint systems, which sometimes contained variables irrelevant to the problem to be solved. BINTEST (Beyleda and Gruhn, 2003) adopted bisection to guide the search process, which, however, might cut the domains of variables that probably contained some solutions. Cadar et al. (2008) proposed a tool named KLEE and employed a variety of constraint solving optimization methods to reach the goal of high coverage. Wang et al. (2013) proposed an interval analysis algorithm using forward dataflow analysis, and adopted Choco (Team, 2010) as the constraint solver. But no matter what techniques are adopted, there are two challenges for the researchers in this field. One is infeasible path detection, without which much of the effort, statistically accounted for 30–75% (Hermadi et al., 2014) of the computation consumption, will become idle. The other is generation time reduction, which is especially important when the

* Corresponding author.
E-mail address: faith.yingxing@gmail.com (Y. Xing).

http://dx.doi.org/10.1016/j.engappai.2015.07.021
0952-1976/

test beds are shifted from small, toy ones to large-scale engineering applications, for the reason that too long generation time is intolerable.

Aiming at constructing a highly automatic test data generation tool that can be used for programs in engineering, we put forward a heuristic method best-first-search branch and bound (BFS-BB) (Xing et al., 2014), adopting branch and bound (BB), which is a classical search algorithm in artificial intelligence. For the purpose of infeasible path detection and generation time reduction as mentioned above, we make improvements on interval arithmetic in BFS-BB, which is used for arc consistency checking. All our work is based on the abstract memory model (AMM) (Tang et al., 2012) in Code Test System (CTS) (http://ctstesting.cn/), which tests real-world programs written in C programming language. AMM underlying automatic test data generation maintains a table of memory states, and the constraints related to the structure of data types can be represented by the table. As for the test data generation method in CTS, the main task is to construct an efficient constraint solver. We take numeric types as an example to describe our method in this paper.

The rest of this paper is organized as follows. Section 2 provides the background and motivation of this paper. In Section 3, we give the theoretical analysis of interval arithmetic and propose the iterative operator to improve it. Two cases are studied in detail to explain the function of the iterative operator in Section 4. In Section 5, we make experimental analyses and empirical evaluations of the proposed method. Section 6 concludes this paper and highlights directions for future research.

2. Background and motivation

As mentioned in Section 1, the problem of path-wise test data generation is in essence a CSP (Kasprzak et al., 2014), where the path refers to a sequence of nodes in a control flow graph (CFG) (McMinn, 2004). To be specific, X is a set of variables $\{x_1, x_2, \ldots, x_n\}$, $D=\{D_1, D_2, \ldots, D_n\}$ is a set of domains, and $D_i \in D$ $(i=1, 2, \ldots, n)$ is a finite set of possible values for x_i. For the path to be covered (denoted as p), D is defined based on the variables' acceptable ranges. One solution to the problem is a set of values to instantiate each variable inside its domain denoted as $\{x_1 \mapsto V_1, x_2 \mapsto V_2, \ldots, x_n \mapsto V_n\}$, $V_i \in D_i$, to make p feasible, meaning that each constraint defined by the PUT along p should be met. It should be noted that one solution is enough for path-wise test data generation, and it is not necessary to try to find all the solutions.

A CSP is generally solved by backtracking search strategies. During the search process, variables are divided into three sets: past variables (short for *PV*, already instantiated), current variable (now being instantiated), and future variables (short for *FV*, not yet instantiated). The idea of the search algorithms is to extend partial solutions. At each step, a variable in *FV* is selected and assigned a value from its domain to extend the current partial solution. It is checked whether such an extension may lead to a possible solution of the CSP and the subtrees containing no solutions based on the current partial solution are pruned.

The techniques for improving a search algorithm are categorized as look-ahead and look-back methods. Look-ahead methods (Frost and Dechter, 1995; Schaerf, 1997) are invoked whenever the search is preparing to extend the current partial solution, and they concern the following problems: (1) how to select the next variable to be instantiated or to be assigned a value; (2) how to select a value to instantiate a variable; (3) how to reduce the search space by maintaining a certain level of consistency. Look-back methods are invoked whenever the search encounters a dead-end and is preparing for backtracking.

The third problem in look-ahead methods is the focus of this paper, which is often solved by local consistency (including node consistency, arc consistency, and path consistency) techniques that associate a CSP with a network of relations, where nodes represent variables and arcs or edges represent constraints. Arc consistency (Cooper et al., 2010; Lecoutre and Prosser, 2006) is the most widely used method, which means that every consistent assignment to a single variable can be consistently extended to a second variable. AC-3 is the simplest arc consistency checking algorithm and is known to be practically efficient (Mackworth, 1977; Wallace, 1993). AC-3 involves a series of tests between pairs of constrained variables. In the enforcement of AC-3, it is not required to process all constraints if only a few domains have changed, and the operations are conducted on a queue of constraints to be processed.

Arc consistency is a basic technique for solving CSPs, but consistency checking algorithms seldom solve CSPs by themselves. Actually, they often assist search algorithms in two ways. One is preprocess before the search starts, and the other is combining with search algorithms to fulfill the search process by reducing the domain of the CSP in question, such as forward checking. Generally, there are arc consistency checking methods within backtracking search algorithms, including BFS-BB as described below.

Xing et al. (2014) described BFS-BB in detail that carries out depth-first search with backtracking, in which the arc consistency checking method interval arithmetic is involved in both preprocess and the search process. Adopting the MC/DC coverage criterion, BFS-BB took nearly 110 min to test the project *aa200c* available at http://www.moshier.net/, which includes 77 functions. The relatively large time consumption is one drawback of BFS-BB. Another drawback of BFS-BB is its inability to detect infeasible paths in advance. According to statistics, there are nearly 34% infeasible paths in *aa200c*. If these paths are considered feasible, there will be large amount of computation wasted in trying to generate test data for them. Fig. 1 shows an example *test 1* to explain why there are infeasible paths that were not detected by BFS-BB. For the sake of easy explanation, we try to generate test data for the path which passes all the *if* statements and finally reaches the print statement. As described in Xing et al. (2014), the set of the domains of all the variables is obtained after the last branch predicate $x2 > 10$, which is $\{x1{:}[-\infty, -11], x2{:}[11, +\infty]\}$. It can be intuitively found that $x1$ is negative and $x2$ is positive, which, however, is in contradiction with the first predicate $x1 > x2$. In other words, the path to be covered is infeasible. But BFS-BB is unable to judge the feasibility of the path in advance due to the conservativeness of interval arithmetic, that is, it sometimes provides much larger intervals. Consequently, the computation consumed on the search will be meaningless. Aiming at solving the two problems with BFS-BB as well as test data generation methods, we made improvements on interval arithmetic based on the analysis in Section 3.

3. Methodology

In this section, we give detailed analysis on interval and interval arithmetic, which is of great importance to BFS-BB, because it is the key part that is in charge of enforcing arc consistency. Based on the analytical result, an iterative operator is proposed.

```
void test1(int x1,int x2){
  if(x1>x2)
    if(x1<-10)
      if(x2>10)
        printf("Reachable?");}
```

Fig. 1. Program *test 1*.