The Journal of Systems and Software 114 (2016) 38–53

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Prioritized static slicing and its application to fault localization

Yiji Zhang*, Raul Santelices

University of Notre Dame, Indiana, USA

ARTICLE INFO

Article history:
Received 2 April 2014
Revised 8 August 2015
Accepted 30 October 2015
Available online 5 November 2015

Keywords:
Static slicing
Probabilistic slicing
Thin slicing
Dependence analysis
Fault localization
Program analysis

ABSTRACT

Static slicing is a popular program analysis used in software engineering to find which parts of a program affect other parts. Unfortunately, static slicing often produces large and imprecise results because of its conservative nature. Dynamic slicing can be an alternative in some cases, but it requires detailed runtime information that can be hard or impossible to obtain or re-create. This is often the case when users report bugs in deployed software. In this paper, we significantly improve the precision of static slicing through PrioSlice, a novel technique that exploits the insight that *not all statements in a static slice are equally likely to affect another statement* such as a failing point. PrioSlice first computes a probabilistic model of the dependencies in the program. In this model, some data dependencies are more likely to occur than others and control dependencies are less likely than data dependencies to propagate effects (e.g., errors). PrioSlice then traverses the program backwards, like static slicing, but in an order defined by the computed dependence probabilities. Our study of fault localization on various Java subjects indicates that PrioSlice can help localize faults *much more effectively* than existing static-slicing approaches.

1. Introduction

Program slicing is a popular program-analysis technique used for many software-engineering tasks, such as fault localization and program comprehension (Weiser, 1984). *Static* program slicing, in particular, finds the set of all statements in the program—the *slice*—that might affect a particular point in that program. Unfortunately, however, static slicing is often too imprecise to be practical because it tends to produce slices whose sizes are very large (Binkley et al., 2007; Santelices et al., 2010), even when considering calling contexts for interprocedural analysis (Horwitz et al., 1990).

To reduce the size of slices and, thus, increase their usefulness, researchers have developed other forms of program slicing such as *dynamic* slicing (Korel and Laski, 1988; Agrawal and Horgan, 1990), variants of it (Beszedes et al., 2002; DeMillo et al., 1996; Hall, 1995), combinations of static slices with execution data (Gupta and Soffa, 1995; Horwitz et al., 2010; Krinke, 2006; Mock et al., 2005), and slice pruning based on various criteria (Acharya and Robinson, 2011; Canfora et al., 1998; Sridharan et al., 2007; Zhang et al., 2006). These variants, however, focus on subsets of all program behaviors and can miss faulty code. Moreover, these techniques can still be imprecise (Santelices et al., 2010).

Dynamic approaches provide concrete insights on how programs behave in typical cases, provided that a *representative* set of executions is used. Unfortunately, such a set of executions is not always available to developers. For fault localization, one failed execution can be enough to find a fault, but multiple executions are often needed to make localization effective (Jones et al., 2002; Jones and Harrold, 2005). In many cases, such as bugs reported by users, developers may not even have one execution to work with. Another problem is that faulty executions can be hard to reproduce for debugging. This problem is exacerbated by non-determinism in software whose behavior depends on external factors such as time and thread interleaving. One possibility is to deploy software instrumented to collect dynamic slices, but this would incur unacceptable runtime overheads. Therefore, when dynamic slicing is not an option, static slicing must be used.

In this paper, we present a novel technique called PrioSlice which considerably increases the effectiveness of static slicing by reducing the negative effects of its imprecision. PrioSlice prioritizes the inspection of a *static slice* (i.e., the subset of the program that affects another program point) using a probabilistic model of how *program dependencies* (Aho et al., 2006; Ferrante et al., 1987) (the building blocks of program slices) occur and how they affect the *slicing criterion* (i.e., a value in a program point from which the slice is computed). To that end, for each statement in a slice, PrioSlice computes a weight in the range [0,1] representing the *likelihood that this statement belongs to the slice*. Thus, PrioSlice tells not only *whether* a statement is in a slice, but also *how much*.

* Corresponding author. Tel.: +1 5748550201.
E-mail addresses: yzhang20@nd.edu, yiji21@gmail.com (Y. Zhang), rsanteli@nd.edu (R. Santelices).

http://dx.doi.org/10.1016/j.jss.2015.10.052
0164-1212/

The probabilistic model of PrioSlice exploits two key insights. The first insight is that not all *data* dependencies (Aho et al., 2006) are equally likely to occur because of control-flow and *aliasing* (Andersen, 1994) reasons. Data dependencies are treated (mostly) as equals by existing static-slicing techniques (Weiser, 1984; Sridharan et al., 2007). Our model and technique, in contrast, gives a greater priority to the data dependencies that are most likely to occur and cause a particular program behavior (e.g., a failure) and, thus, more quickly identifies its causes (e.g., a fault).

The second insight used in this model is that *control* dependencies (Ferrante et al., 1987) are usually weaker than data dependencies at propagating effects such as errors (Masri and Podgurski, 2009; Santelices et al., 2010; Sridharan et al., 2007). Therefore, they should not be treated as equals with data dependencies for slice inspection, as Weiser's method does (Weiser, 1984), but they should not be discarded upfront either as, for example, *thin* slicing does (Sridharan et al., 2007). Our model incorporates both types of dependencies while giving them different weights according to their respective effect-propagation potential estimated by our probabilistic analysis.

Concretely, PrioSlice first computes the static backward slice from a slicing criterion in a program and then solves the system of equations given by the model for the dependence graph of that program. The solution for these equations is the set of weights for the statements in the static slice. These weights are the result of using a standard method for solving real-valued data-flow problems (Ramalingam, 1996). PrioSlice uses these results to perform a *best-first* traversal (Pearl, 1984) of the slice, which replaces Weiser's breadth-first traversal approach (Weiser, 1984) of static dependencies by prioritizing statements according to their weight.

Unlike statistical approaches (e.g., Jones and Harrold, 2005) which provide *seemingly-random orders* to inspect programs (Parnin and Orso, 2011), static-slicing approaches such as PrioSlice *navigate the dependence graph in order* by visiting only neighbors of visited nodes as suggested by Weiser (1984). Each new statement to inspect is reached via a direct dependence to another statement already inspected. Navigating dependencies is, thus, a more natural way for developers to inspect programs (Parnin and Orso, 2011).

To study the feasibility and effectiveness of PrioSlice, in this paper, we chose *fault localization* as the application—arguably the most important one. Fault localization is the first step for debugging a program, in which the causes of a reported failure must be found before the program can be fixed. We implemented our technique for slicing Java-bytecode programs using our dependence-analysis infrastructure DUA-Forensics (Santelices and Harrold, 2007; Santelices et al., 2013a). Then, we applied three static-slicing techniques to several faults across various Java subjects. These techniques are PrioSlice, Weiser's traversal approach for static slicing (Weiser, 1984), and thin slicing (Sridharan et al., 2007).

Our results for these subjects and faults indicate with statistical significance that PrioSlice can reach the faulty code faster than static slicing. PrioSlice traverses, on average, about 13% of the program, whereas Weiser's approach requires an average inspection of about 27% of the program to find each fault. Meanwhile, thin slicing, when it was able to find a fault at all (only 12 out of 60 in total),[1] also required to inspect about 18% of the program on average. For those faults, PrioSlice required about the same amount of program (i.e. 17%) to be inspected on average. Though we can not draw the conclusion that PrioSlice is significantly better than thin slicing, we also can't tell that thin slicing is different from PrioSlice. (Not to mention that PrioSlice was applicable to all faults, whereas thin slicing was only able to 20% of the faults.)

[1] Control dependencies can be added for the remaining faults after data dependencies are exhausted, but the order in which this is done is unclear (Sridharan et al., 2007).

The most important benefit of this work is that it provides a new way of looking at static slices in which statements are distinguished by relevance rather than just by membership in the slice. This feature makes PrioSlice improve fault localization considerably. Its focused traversal of dependence graphs also hints at its potential for related applications such as program comprehension. We also applied a preliminary *forward* version of this model (Santelices and Harrold, 2010; Santelices et al., 2013b) for change-impact analysis with promising first results. Finally, the probabilistic model underlying PrioSlice still has room for improvements so that its benefits can further increase in the future.

In all, the contributions of this paper include:

- A new form of static program slicing that indicates not only whether, but also how much, each statement belongs to a slice.
- A new technique, PrioSlice that realizes this concept of probabilistic slicing to prioritize the traversal of slices by the estimated degree of membership of their statements.
- A study indicating that PrioSlice can considerably reduce the effort of static fault localization with respect to Weiser's original approach and thin slicing.

2. Background

This section presents core concepts needed for this paper and illustrates those concepts using the example program of Fig. 1.

2.1. Program dependencies

Control and data dependencies are the building blocks of program slicing (Weiser, 1984; Horwitz et al., 1990). A statement T is *control dependent* (Ferrante et al., 1987) on a statement S, denoted (S, T), if a control-flow (jump) decision taken at S determines whether T is necessarily executed. For example, in Fig. 1, line 4 is control dependent on line 3 because the decision made at 3 determines whether 4 is necessarily executed.

A statement U is *data dependent* (Aho et al., 2006) on a statement D if a variable v can be defined (written) at D and used (read) at U and there is a *definition-clear path* in the program for v (i.e., a path that does not re-define v) from D to U. This definition of data dependence implies that, if v is accessed via a pointer or reference p, p points to the same memory location at D and U. We denote a data dependence of U on D for variable v as (D, U, v), or just (D, U).

2.2. Program slicing

Program slicing was originally developed by Weiser (1984). A *static slice* for a set of variables V at a program point C—the *slicing criterion*—is the subset of all program statements that affect those values at that point. A slice can also be defined simply for a program point C as the set of all statements that affect either the values used at C or the execution of C.

To realize Weiser's approach, we use the *dependence graph* of the program. This graph is using the data and control dependencies reachable backwards from the slicing criterion, where nodes are statements and edges are dependencies. Weiser's approach performs a backward breadth-first traversal of this graph to produce a *visit order* of the statements. For fault localization, this order can be used to measure the *effort* required to find a fault—the number of nodes traversed until the fault is reached.

To illustrate, consider the example program in Fig. 1. Suppose that we want to determine all statements in this program that affect the value of v at statement 16. Using the backward transitive closure of control and data dependencies from that statement, we obtain the backward static slice $\langle 1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 13\rangle$.