The Journal of Systems and Software 129 (2017) 60–78

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Observational slicing based on visual semantics

Shin Yoo[a], David Binkley[b,*], Roger Eastman[b]

[a] School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea
[b] Department of Computer Science, Loyola University Maryland, Baltimore, MD, USA

ARTICLE INFO

Article history:
Received 2 February 2015
Revised 26 January 2016
Accepted 6 April 2016
Available online 13 April 2016

Keywords:
Observation
Non-traditional semantics

ABSTRACT

Program slicing has seen a plethora of applications and variations since its introduction over 35 years ago. The dominant method for computing slices involves significant complex source-code analysis to model the dependencies in the code. A recently introduced alternative, observation-based slicing, sidesteps this complexity by observing the behavior of candidate slices. Observation-based slicing has several other strengths, including the ability to easily slice multi-language systems.

However, the initial implementation of observation-based slicing, ORBS, remains rooted in tradition as it captures semantics by comparing sequences of values. This raises the question of whether it is possible to extend slicing beyond its traditional semantic roots. A few existing projects have attempted this but the extension requires considerable effort.

If it is possible to build on the ORBS platform to more easily generalize slicing to languages with non-traditional semantics, then there is the potential to vastly increase the range of programming languages to which slicing can be applied. ORBS supports this by reducing the problem to that of generalizing how semantics are captured. Taking Picture Description Languages as a case study, the challenges and effectiveness of such a generalization are considered. The results show that not only is it possible to generalize the ORBS implementation, but the resulting slicer is quite effective, removing from 8% to 98% of the original source code with an average of 83%. Finally a qualitative look at the slices finds the technique very effective, at times producing minimal slices.

1. Introduction

At the time of its introduction program slicing was devised for use with simple imperative source code (Weiser, 1979). During the ensuing 35 years the applicability of the technique has been expanded to an ever widening definition of source code (Harman, 2010). Examples include slicing object-oriented code (Larsen and Harrold, 1996), slicing binary executables (Cifuentes and Fraboulet, 1997), and slicing finite-state models (Androutsopoulos et al., 2011).

Informally, Weiser defined a slice as a subset of a program that preserves the behavior of a specific computation from the program. Slicing allows one to find semantically meaningful decompositions of a program. For example, it allows the tax computation to be extracted from a mortgage payment system. Weiser's definition of a slice includes two requirements: a syntactic requirement and a semantic requirement. The syntactic requirement is that the slice be obtainable from the original program by deleting elements (typically statements). Relaxing this requirement has been helpful in slicing programs with unstructured control flow (Choi and Ferrante, 1994; Harman et al., 2006) and led to the development of Amorphous Slicing (Harman and Danicic, 1997; Harman et al., 2003).

The semantic requirement defines the behavior of a slice. It requires that a slice capture a subset of the original program's semantics. For a single threaded, single procedure imperative program this can be done using the sequence of values produced at each program point (Weiser, 1979). Generalization to sets of sequences-of-values can capture the semantics of more complex programs such as those with procedures (Binkley, 1993; Horwitz et al., 1990) threads (Krinke, 1998), and objects (Larsen and Harrold, 1996).

Recently *observation-based slicing* (Binkley et al., 2014; 2013) was introduced to tackle two long-standing challenges in program slicing: slicing multi-language systems and slicing systems that contain (third party) components whose source code is often not available. Observation-based slicing works by observing the semantics of candidate slices. This approach supports a generalization of program slicing to a broader range of source code kinds including languages with *non-traditional semantics* (i.e., where the meaning of a program is not captured by sequences of values).

* Corresponding author.
E-mail addresses: shin.yoo@kaist.ac.kr (S. Yoo), binkley@cs.loyola.edu (D. Binkley), reastman@loyola.edu (R. Eastman).

http://dx.doi.org/10.1016/j.jss.2016.04.009
0164-1212/© 2016 Elsevier Inc. All rights reserved.

This paper explores the generalization by building on previous work presented at SCAM 2014 (Yoo et al., 2014). It considers, as a representative example of languages with non-traditional semantics, Picture Description Languages (PDLs). Source code written in such a language specifies a graphic image in terms of objects such as shapes, boxes, arrows, etc. These languages can be Turing-complete, or focused on output description with limited control structures. Examples of such languages include Postscript, pic, xfig, html (including code written in embedded languages such as CSS and JavaScript) and TikZ/PGF. While informally the semantics of such languages can be straightforward, requiring only visual inspection, the problem of slicing them is subtle as discussed in Section 4.

While slicing languages with non-traditional semantics is, in itself, an interesting problem, there are also practical motivations behind the proposed technique. First, slicing PDLs can help users understand how to generate (i.e., write the code for) complicated diagrams. Users of PDLs often rely on online repositories (or *galleries*) of various diagrams to learn how to program specific shapes and layouts. In this context, slicing can serve as a program comprehension aid where users can select specific parts of a larger diagram and allow the slicer to identify the PDL statements responsible for generating the selected parts. Second, slicing PDLs can help locate software faults that manifest themselves visually, such as HTML presentation failures (Mahajan and Halfond, 2015). Given that dynamic web pages usually involve multiple languages such as HTML, CSS, and JavaScript, the observation-based nature of the proposed slicing technique is a significant benefit, as it can easily handle multiple language descriptions.

By taking on the challenge of slicing languages whose output is visual rather than those that can be captured using more traditional semantics, such as Weiser's sequences of values, this work shows that it is possible to increase the variety of languages to which program slicing can be applied. More specifically, the two main contributions of this paper are:

- a generalization of observation-based slicing to languages with non-traditional semantics, and
- an empirical study that demonstrates the application and operation of this new approach, using PDLs as representative examples.

The research questions used to investigate the generalization are introduced in Section 3 followed by the generalization itself in Section 4. The empirical investigation begins in Section 5 with the study of an initial implementation built using off-the-shelf components and experiments investigating its quantitative and qualitative aspects. This initial study uncovers several shortcomings, discussed in Section 6, which leads to an improved implementation. Section 7 empirically investigates the performance of the improved implementation. Before these studies, a review of program slicing and specifically the observation-based approach is given in Section 2. Finally, the paper ends with a discussion of related work, future work, and a brief summary.

2. Program slicing

Program slicing has many applications, including testing (Binkley, 1998; Hierons et al., 2002), debugging (Kusumoto et al., 2002; Weiser and Lyle, 1985), maintenance (Gallagher and Lyle, 1991; Hajnal and Forgács, 2011), re-engineering (Cifuentes and Fraboulet, 1997), re-use (Beck and Eichmann, 1993; Cimitile et al., 1995), comprehension (De Lucia et al., 1996; Tonella, 2003) and refactoring (Ettinger and Verbaere, 2004). A more complete introduction can be found in several surveys and tutorials such as Gallagher and Binkley's Foundation of Software Maintenance article (Binkley and Gallagher, 1996).

Slicing can be classified as either static or dynamic: a static slice (Weiser, 1982) of program P is a subset of P that has the same behavior as P for a specified variable at a specified location (a slicing criterion) *for all possible inputs*, while a dynamic slice (Korel and Laski, 1988) preserves this behavior for only a single input (or a small set of inputs).

Weiser's original definition of a static slice, used the state trajectory projection function, PROJ_C (Weiser, 1982), which projects out of a trajectory T those elements relevant to slicing criteria C. A trajectory is a record of the values computed by a program (e.g., the sequence of values assigned to the left-hand-side variable in an assignment statement). For static slicing the slicing criteria $C = (v, l)$ includes a variable v and a line (location) l from the source code. The criterion for a dynamic slice, denoted $(v, l, \mathcal{I})$, adds a set of inputs $\mathcal{I}$ (a variant replaces v with v_i, the ith occurrences of v in the trajectory).

Most static and dynamic slicing algorithms employ complex dependence analysis to extract information from a program (and its execution in the case of dynamic slicing). These algorithms then decide which statements should be retained to form the slice. The recently introduced observation-based slicing (Binkley et al., 2014; 2013) replaces the complex and expensive dependence analysis with observation. Its first implementation, ORBS, computes a slice by *deleting* statements, *executing* the candidate slice, and *observing* its behavior. The use of execution makes the approach inherently dynamic in nature. It also means that ORBS takes a very operational view of program semantics. One advantage of this view is that observation is considerably simpler to work with than the complex construction of a semantic model capturing dependence (Podgurski and Clarke, 1990; Parsons-Selke, 1989). For ORBS all that is required is an algorithm for comparing projected executions. Thus ORBS replaces the complexity of generating a correct answerer with the simpler task of testing correctness.

Being freed from complex program dependence analysis allows observation-based slicing to focus on subsets of a program; thus an observation-based slice further extends the slice criteria to include *components of interest, CoI*. Slicing's deletion is restricted to the *CoI*. This enables, for example, slicing programs that contain binary components and source code such as third-party libraries, which are excluded from *CoI* and thus need not be changed by the slicer. Consequently, an observation-based slice, taken with respect to the criteria $(v, l, \mathcal{I}, CoI)$, preserves the state trajectory for v at l for the selected inputs in $\mathcal{I}$, while deleting statements from the components of *CoI* but no other components.

Furthermore, observation-based slicing is inheritantly language-independent. It achieves this by replacing the deletion of statements (a language specific concept) with the deletion of lines of text. While no assumption about the contents of a line is made (e.g., ORBS does not assume that the source files are formatted with one statement per line) slice quality degrades if multiple statements occupy the same line as they are inseparable at the lexical level. More formally, an ORBS slice is defined as follows:

Observation-based slicing (Binkley et al., 2014): An *observation-based* slice S of program P taken with respect to slicing criterion $C = (v, l, \mathcal{I}, CoI)$ composed of variable v, line l, set of inputs $\mathcal{I}$, and components of interest *CoI*, is any executable program with the following properties:

1. S can be obtained from P by deleting zero or more lines from *CoI*.
2. Whenever P halts on input $I \in \mathcal{I}$ with state trajectory $T(P, I, v, l)$ then S also halts on input I with state trajectory $T(S, I, v, l)$ such that $\mathsf{PROJ}_C(T(P, I, v, l)) = \mathsf{PROJ}_C(T(S, I, v, l))$.

The key to observation-based slicing is *observing* the behavior of *candidate slices*. The initial ORBS implementation forms candidate slices by deleting a continuous sequence of lines from the current