Seeing is Slicing: Observation Based Slicing of Picture Description Languages

Shin Yoo
Department of Computer Science
University College London
London, UK
Email: shin.yoo@ucl.ac.uk

David Binkley
Department of Computer Science
Loyola University
Maryland, Baltimore, USA
Email: binkley@cs.loyola.edu

Roger Eastman
Department of Computer Science
Loyola University
Maryland, Baltimore, USA
Email: reastman@loyola.edu

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

However, 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 algorithm, but the resulting slicer is quite effective removing from 27% to 98% of the original source code with an average of 85%. Finally a qualitative look at the slices finds the technique very effective, at times producing minimal slices.

I. INTRODUCTION

At the time of its introduction program slicing was devised for use with simple imperative source code [1]. During the ensuing thirty-five years the applicability of the technique has been expanded to an ever widening definition of source code. Examples include slicing object-oriented code [2], slicing binary executables [3], and slicing finite-state models [4].

Informally, Weiser defined a slice as a subset of a program that preserves the behavior of the program for a specific computation. 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 [5], [6] and led to the development of Amorphous Slicing [7], [8].

The semantic requirement defines the behavior of a slice. It requires that the 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 [1]. Generalization to sets of sequences-of-values can capture the semantics of more complex programs such as those with procedures [9], [10] threads [11], and objected-oriented code [2].

Recently *Observation-Based Slicing* (ORBS) [12], [13] 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. ORBS 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).

This paper explores the generalization. To do so it considers, as representative examples 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 boxes, arrows, etc. Examples of such languages include Postscript, pic, xfig, and TikZ/PGF. While informally the semantics of such languages is straightforward, requiring only visual inspection, the problem of slicing them is subtle as discussed in Section IV-A.

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 that treats 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 generalization, an initial algorithm, and experiments investigating its quantitative and qualitative aspects, are presented in Section IV. Before this section a review to program slicing and specifically the ORBS approach is given in Section II. Finally the paper ends with a discussion of related work, future work, and a brief summary.

II. PROGRAM SLICING

Program slicing has many applications, including testing [14], [15], debugging [16], [17], maintenance [18], [19], re-engineering [20], re-use [21], [22], comprehension [23], [24] and refactoring [25]. A more complete introduction can be found in several surveys and tutorials such as Gallagher and Binkley's Foundation of Software Maintenance article [26].

Slicing can be classified as either static or dynamic: a static slice [27] 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 [28] 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 [27], 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 [12], [13] replaces the complex and expensive dependence analysis with observation. In more detail, ORBS computes a slice by *deleting* statements, *executing* the candidate slice, and *observing* its behavior. In doing so 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 [29], [30]. For ORBS all that is required is an algorithm for comparing projected executions.

Being freed from complex program dependence analysis allows ORBS to focus on subsets of a program; thus an ORBS 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. Thus, an ORBS 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, ORBS is 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:

ORBS Slice [12]: 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 and produces 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 ORBS is *observing* the behavior of *candidate slices*. A candidate is formed by deleting a continuous sequences of lines from the current slice. It is then validated using compilation and execution. If the candidate fails to compile, it cannot produce the correct trajectory and is thus rejected. Similarly, if the candidate produces a different projected trajectory than the original program, then the candidate is rejected. Otherwise, if it passes both checks, the candidate is accepted as the current slice. The ORBS algorithm systematically forms candidates until no more lines can be deleted.

Operationally, the ORBS implementation produces the trajectory $T(P, I, v, l)$ by injecting, just before line l, a (necessarily language specific) statement that writes the value of v to a file. ORBS is then able to leverage the existing tool chain to build and execute the program. Doing so avoids the costly development of language-specific program analysis tools.

An ORBS prototype [12] was able to successfully slice programs known to be a challenge for tradition dependence-based slicers such as the program adorning the 2001 SCAM Mug [31]. Furthermore, its slices compare favorably with those created by similar techniques such as Critical Slicing [32] and several slicing variants based on Delta Debugging [33]. Finally, slices of bash, which includes 118,167 source lines of code (SLOC) [34] written in eight different languages, include between 10% to 17% of the original program.

III. RESEARCH QUESTIONS

The following research questions are used to study the generalization of ORBS to languages with non-standard semantics, specifically PDLs.

RQ$_1$: What are the impacts of generalizing ORBS to PDLs given their non-traditional semantics?

This first research question is aimed at gaining knowledge. It investigates the modifications to both the *definition* and the *implementation* of ORBS necessary to provide an effective slicing implementation for picture description languages.

RQ$_2$: How much reduction is achieved by slicing?

A key goal of slicing is to remove (slice away) unwanted parts of the source code. RQ$_2$ considers the effectiveness of the slicer at doing so for PDLs. It does this by considering five slices of each of the five picture descriptions shown in Table I.

RQ$_3$: What is the visual precision of the resulting slices?

RQ$_3$ considers the subjective correctness of the slices. Because the languages being sliced describe pictures, it is reasonable to consider whether the slices appear visually correct. Thus the third research question considers the subjective visual correctness and precision of the new slicer.

TABLE I.   STUDIED PICTURE DESCRIPTIONS

Name	Language	LoC
cone	TikZ/PGF	74
hydrogen	TikZ/PGF	61
raindrop	TikZ/PGF	44
shapes	TikZ/PGF	25
slice	Pic	262

IV.   EMPIRICAL INVESTIGATION

A. RQ_1 – Impacts of the Generalization

RQ_1 addresses two challenges in slicing PDLs: generalizing the definition of a slice and then in extending ORBS. The most significant questions here concern the feasibility of defining the slicing criterion and of capturing the semantics of languages with non-traditional semantics.

The generalization involves modifying the definition of the "slicing criteria" and re-envisioning how to capture the projected semantics. The traditional slicing criteria includes a variable v, a line number l, and a set of inputs $\mathcal{I}$. This definition could be directly ported to slicing picture descriptions: however, as their output is visual, a visual slicing criteria seems preferable. Such a criteria takes the form of an image cropped from an original image. (Reversing this observation, a similar notion is possible with traditional slicing where the slicing criteria would be some subset of the program's output. However, this notion seems more fitting when that output is visual.) Thus the goal of slicing goes from preserving the behavior of v at l for inputs in $\mathcal{I}$ to preserving the visual appearance of a set of template images, $\mathcal{T}$. For presentation simplicity the rest of the paper (e.g., Algorithms 1 and 2) assumes that the sets $\mathcal{I}$ and $\mathcal{T}$ are a singleton sets.

Turning to the semantics, as long as it is possible to do something equivalent to *observing* the behavior of the program and *comparing* the projected behaviors of a candidate slice and the original program, then it is possible to capture the semantic requirement of a slice. For images, the comparison must check if the slicing criterion (the cropped image) is present in the rendered version of a candidate slice. This is a problem known as *template matching* [35]: given a source image I and a smaller template image T, the goal of template matching is to detect the area of I that best matches T. When T is clipped from I then in principle there is a perfect match for T in I.

To better understand the impact of these generalizations, it is necessary to consider their implementation. The algorithm for slicing picture descriptions is a modification of the original ORBS algorithm for slicing traditional source code. The core of both algorithms is shown as Algorithms 1 and 2. As described in the previous section, ORBS, begins by annotating (instrumenting) the program to be sliced and then capturing the initial trajectory in the variable T (Lines 1 and 2 of Algorithm 1). The main loop of the algorithm then repeatedly creates candidate slices, S', by deleting up to five consecutive lines starting from each line in the current slice[1]. The candidate slice replaces S as the current slice on Line 10 if S' compiles and produces the correct projected trajectory.

---

[1]While other approaches are also possible, the linear and consecutive deletion approach, called *moving deletion window*, is preferred for performance reasons. See the original ORBS algorithm for more detail [12].

Algorithm 1: ORBS [12]

ORBSLICE(P, C)
Input: Program P, slicing criterion C
Output: A slice S of P for C
(1) $S \leftarrow$ INSTRUMENT(P, C)
(2) $T \leftarrow$ EXECUTE(COMPILE(S), $C.\mathcal{I}$)
(3)   ...
(4)   **repeat**
(5)   ...
(6) $S' \leftarrow candidate\_slice(S)$
(7)   **if** COMPILE(S') $= success$
(8) $T' \leftarrow$ EXECUTE(COMPILE(S'), $C.\mathcal{I}$)
(9)   **if** $PROJ_C(T) = PROJ_C(T')$
(10) $S \leftarrow S'$
(11)   ...
(12)   **until** $nochanges$
(13)   **return** S

Algorithm 2: VORBS

VORBSLICE(P, C)
Input: Picture description P, slicing criterion C
Output: A slice S of P for C
(1) $S \leftarrow P$
(2) $R_0 \leftarrow$ MATCH(COMPILE(S), $C.\mathcal{T}$)
(3)   ...
(4)   **repeat**
(5)   ...
(6) $S' \leftarrow candidate\_slice(S)$
(7)   **if** COMPILE(S') $= success$
(8) $R \leftarrow$ MATCH(COMPILE(S'), $C.\mathcal{T}$)
(9)   **if** $R <= R_0$
(10) $S \leftarrow S'$
(11)   ...
(12)   **until** $nochanges$
(13)   **return** S

Algorithm 2, VORBS (*visual ORBS*), slices PDLs. This algorithm replaces the computation and comparison of trajectories with the computation and comparison of *template-matching scores*. There are a range of template matching algorithms. The current implementation uses the *normalized sum of square differences* (SSD) metric to compare the target template image T to an image I. SSD is a well known and widely studied metric for image registration with general applicability, and is effectively a least squares minimization for its translation parameters (x, y). Efficient off-the-shelf implementations are available as in the OpenCV version used in the implementation of VORBS. As a least squares minimization, the SSD metric is robust to a linear transformation $T = gI + b$ between image intensity values that might be corrupted by Gaussian noise and performs as well as optimal matched filter correlation in straightforward image registration.

Whether the image registration is straightforward depends on the consistency of the image rendering pipeline used to generate the original image, the template, and the sliced image [35]. The use of pdflatex and groff to generate the vectorized diagrams and SIPS to generate bitmaps for the registration, provides a very stable pipeline where the resulting images are very consistent at the pixel level. This consistency means that any efficient image registration algorithm would

likely be effective to the requirement of estimating translation to one pixel, including absolute difference.

The specific implementation used in the experiments is the normalized squared difference implemented in OpenCV under the option CV_TM_SQDIFF_NORMED. This algorithm matches template T in image I by computing the following score, R, for each point (x, y) in image I:

$$R(x, y) = \frac{\sum_{x'y'} \left(T(x', y') - I(x + x', y + y')\right)^2}{\sqrt{\sum_{x'y'} T(x', y')^2 \sum_{x',y'} I(x + x', y + y')^2}}$$

This computation is applied to rasterized bitmapped images.

The best-case match occurs when template T is cropped from I. Here, in theory, $R(x, y)$ is zero at the (x, y) point of I from which T was cropped. However, in practice, rasterizing a vector graphic and potentially the approximations made in various lossy image formats can lead to non-zero R values. To partially compensate for this, the R value computed from the original image and the slicing criteria is used as a threshold, R_0 (Line 2 of Algorithm 2). Subsequently, if a candidate slice produces an R value no more than R_0 (Line 9), then the candidate becomes the current slice (Line 10).

VORBS is implemented in python (version 2.7.5) using SIPS (Scriptable Image Processing System, version 10.4.4) [36] to rasterize vector graphic into the JPEG format. It also uses the OpenCV library (version 2.4.9) [37] to perform the template matching, and finally, the cropping was performed directly on PDF images using Preview.app (version 7.0) on a Mac OS X (version 10.9.2). For TikZ/PGF source code, pdflatex (version 2.5-1.40.14) from TeXLive 2013 is used to build the PDF, while to build from pic source code, GNU pic and groff (version 1.19.2) are used to generate postscript, which is subsequently converted to PDF using the utility ps2pdf.

In summary for RQ_1, the impacts of generalizing ORBS to slicing PDLs is a modification of the slicing criteria to use images cropped from the original image and a re-envisioning how to capture the projected semantics using template matching. The next two subsections consider the quantitative and qualitative behavior of the generalized algorithm.

B. RQ_2 – Reduction Achieved

Turning to RQ_2, the reduction achieved by VORBS is considered. The investigation considers two picture description languages: TikZ/PGF [38] and pic [39]. PGF and TikZ are both invoked as TeX macros, with PGF being the low-level language and TikZ a collection of high-level macros built over PGF. The pic language is a procedural language with macros, branches, and loops.

The experiments consider four TikZ/PGF diagrams, cone, hydrogen, raindrop, and shapes, taken from the public on-line repository at http://www.texample.net/tikz/examples. The rendered versions of these four picture descriptions are shown in Figure 1. The rendered pic image, shown in the top of Figure 2, is an example taken from a paper describing how to perform program slicing using the System Dependence Graph [10]. Table I summarizes the five subjects.

For each picture, five sub-images were cropped out and used as slicing criteria, producing a collection of 25 slices in

TABLE II.   PICTURE DESCRIPTIONS SLICE STATISTICS

Subject	Lines in CoI	Slicer Passes	Deleted Lines	Percent Reduction
pdg-figure-1	262	5	251	96%
pdg-figure-2	262	4	258	98%
pdg-figure-3	262	3	249	95%
pdg-figure-4	262	3	242	92%
pdg-figure-5	262	3	245	94%
cone-1	74	4	34	46%
cone-2	74	3	37	50%
cone-3	74	3	71	96%
cone-4	74	3	68	92%
cone-5	74	3	62	84%
hydrogen-1	61	3	50	82%
hydrogen-2	61	3	54	89%
hydrogen-3	61	3	54	89%
hydrogen-4	61	5	51	84%
hydrogen-5	61	3	57	93%
raindrop-1	45	4	28	62%
raindrop-2	45	4	33	73%
raindrop-3	45	3	22	49%
raindrop-4	45	4	12	27%
raindrop-5	45	4	39	87%
shapes-1	25	3	9	36%
shapes-2	25	2	11	44%
shapes-3	25	3	17	68%
shapes-4	25	2	7	28%
shapes-5	25	3	14	56%
Average		3.3		85%

total. These sub-images, some of which are shown in Figures 2 and 3, cover a range from simple to complex portions of the original images[2]. The 25 slices are summarized in Table II, which shows the number of lines in the CoI of the original image's source code, the number of passes the slicer used, the total number of lines deleted, and finally the percent size reduction in the source code. The number of passes is the number of iterations of the slicer's outermost loop (Line 4 in both algorithms). Each pass considers the deletion of each statement in the current slice and a limited number of its subsequent statements. Over all 25 slices the range of reductions is wide, but as seen in the weighted average, the overall reduction, 85%, is substantial.

The percent reduction is given as a percentage of the CoI. In the experiments the CoI is the file that produced the image being sliced. For TikZ code this excludes a 23 line LaTeX harness that establishes the document class and includes necessary packages. The pic code needs no such wrapper, but the CoI excludes the other 33 files used to build the paper from which the image was taken.

In summary, for RQ_2 the case study of 25 slices shows that VORBS can be used to compute slices of multi-language picture descriptions and that the resulting slices are significantly smaller than the original. Pragmatically, this means the VORBS can be used to automatically extract, from a picture description, code that corresponds to a particular sub-image.

---

[2]The complete set of PDL source code and slicing criteria images, as well as the resulting slices (the source code and rendered versions), can be found at http://www0.cs.ucl.ac.uk/staff/s.yoo/vorbs/index.html.
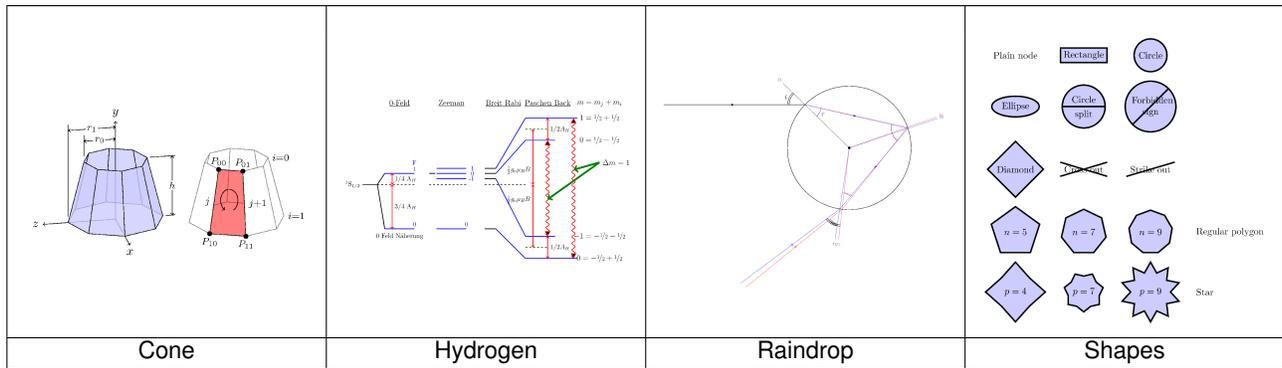
Fig. 1. The four TikZ/PGF images sliced

C. RQ3 – Visual Comparison

Finally, while RQ_2 took a quantitative look at the slices, RQ_3 considers the slices qualitatively. It looks at the resulting slices both in their source form and in their rendered form. The examination includes the slice with the largest reduction, pdg-figure-2 and that with the smallest reduction, raindrop-4.

Figure 2 shows the example pic image, which was taken from a paper on how to perform program slicing using the System Dependence Graph [10]. The image shows a sample sliced program and its dependence graph. The source code for this image is written primarily in pic. The image also includes labels typeset using eqn and some limited troff markup.

Below the original image three slices are shown (slices pdg-figure-2 – pdg-figure-4 from Table II). For each slice the first column shows the sub-image clipped from the original and used as the slicing criteria. The middle column is the slice and the third column is the image rendered from the slice.

The first example slice is taken with respect to the image of the source code shown in the upper left of the original image. As seen in the third column, when rendered the resulting slice produces exactly the desired output. The actual slice, shown in the center column, includes the minimal subset of the original program necessary to produce the correct rendered output.

The second slice is taken with respect to the sub-image that contains the dependence graph's entry vertex. The slicing criteria was deliberately clipped from the original image to omit the edges incident on this vertex. As with the first slice, the second slice, shown in the center column, is minimal. Its four lines start a picture (.PS), update the default height and width for an ellipse, and then finally draw the ellipse. Note that pic is very forgiving and produces the desired output even though the input is without a picture end (.PE) directive.

The final slice is more complex in part because the source code for the image of the while vertex is in the middle of the original code. In contrast, the source code for the entry vertex was near the beginning. The slice includes equation typesetting (appearing between at signs) and picture drawing elements. Although it is not visually evident nor immediately obvious from the source code, the slice is actually minimal.

To see this, consider first the last line of the slice, which is shown wrapped around in the figure. This line draws the ellipse from which the slicing criterion was clipped. This code makes direct use of the variables ellipsewid and xgap; thus their assignments are retained in the slice. Furthermore it

implicitly uses ellipseht requiring the inclusion of its definition. The inclusion of "**Program *Main***" was initially a surprise, as this eqn line seems unrelated to the pic code. The connection comes from the inclusion of "@gsize 9@" which sets eqn's global font size to 9 point. Because this is not the default size, deletion of this line changes the appearance of the labels in the vertices of the image, specifically the label of the while vertex; thus this line setting the font size must be retained in the slice. The first three lines change eqn's hot character from the default to the at symbol, @, and thus are required for a reason similar that of the "gsize" directive. Inclusion of these first four lines is an excellent example of the power brought by the use of observation. The VORBS slicer is able to capture dependences that are not explicit in the code and, therefore, hard to capture in a formal model.

Finally, the code that draws the other two ellipses is needed because their position determines the position of the while vertex. The pic description of an element has the syntax

<name>: <entity> "<label>" <location>

where a location can be relative to other elements. In this case the while vertex has the location "at l1 + (xgap,0)". This is a use of the name "l1" requiring the inclusion of the element defining l1. Transitively, this then requires the inclusion of ygap and the line defining the name "Entry".

The second set of three slices, shown in Figure 3, are of the image Raindrop, which is shown in Figure 1. The source for this figure is shown in Figure 4 along with one of its slices. The slices of this image (slices raindrop-2 – raindrop-4 from Table II) illustrate the language independence of VORBS as the same slicer constructed these slices and those in Figure 2.

Considering each slice in turn, the first slice is taken with respect to the line with the arrow in the upper left of the original figure. The slice successfully removes most of the original figure. In this case a minimal slice would also omit the diagonal line segment labeled n. In the source shown in Figure 4, this line segment is the third \draw command and the line with the arrow does not depend on it. However, its removal shifts the remaining figure up. At first glance it appears that this should not impact the computation of R but it does. A preliminary investigation of this phenomena (see Section VI) suggests that subpixel movement causes subpixel interpolation errors at pixel frequencies. These errors introduce blurring, which modifies R's value.

The second slice is taken with respect to the dot at the center of the figure and its incident lines. The resulting slice

Original Image		

program *Main*
 i := 1
 while *i* < 11 **do**
  i := *i* + 1
 od end(*i*)

Edge Key
—————— control dependence
————▸ loop-independent flow dependence
—+—▸ loop-carried flow dependence
- - - ▸ def-order dependence

Figure <|pdg_slice_figure|>. The graph and the corresponding program that result from slicing the program dependence graph from Figure 1 with respect to the final-use vertex for *i*.

Slicing Criteria	Slice	Rendered Slice

program *Main*
 i := 1
 while *i* < 11 **do**
  i := *i* + 1
 od end(*i*)
	.EQ
	delim @@
	.EN
	@gsize 9@\fBprogram\fP @Main@ ①
	 @i := 1@ ②
	  \fBwhile\fP @i˜<˜11@ \fBdo\fP ③
	   @i := i + 1@ ④
	  \fBod\fP ⑤
	 \fBend\fP(@i@) ⑥
		program *Main*①
		 i := 1 ②
		 while *i* < 11 **do**③
		  i := *i* + 1④
		 ⑤**od end**(*i*)⑥

(ENTRY)	.PS ellipseht = ellipseht*.6 ellipsewid = ellipsewid*1.4 Entry: ellipse "ENTRY" ①	(ENTRY)①
(while *i* < 11)	.EQ delim @@ .EN @gsize 9@\fBprogram\fP @Main@ ① .PS xgap = 1.5 ygap = .75 ellipseht = ellipseht*.6 ellipsewid = ellipsewid*1.4 Entry: ellipse "ENTRY" ② ellipsewid = ellipsewid*.7 I1: ellipse "@i := 1@" at Entry - (.5*xgap, ygap) ③ While1: ellipse wid ellipsewid*1.4 + "\fBwhile\fP @i˜<˜11@" at I1 + (xgap, 0) ④	**program** *Main* ① (ENTRY)② (i := 1)③ (while *i* < 11)④

Fig. 2. Three slices of a picture written using pic and eqn.
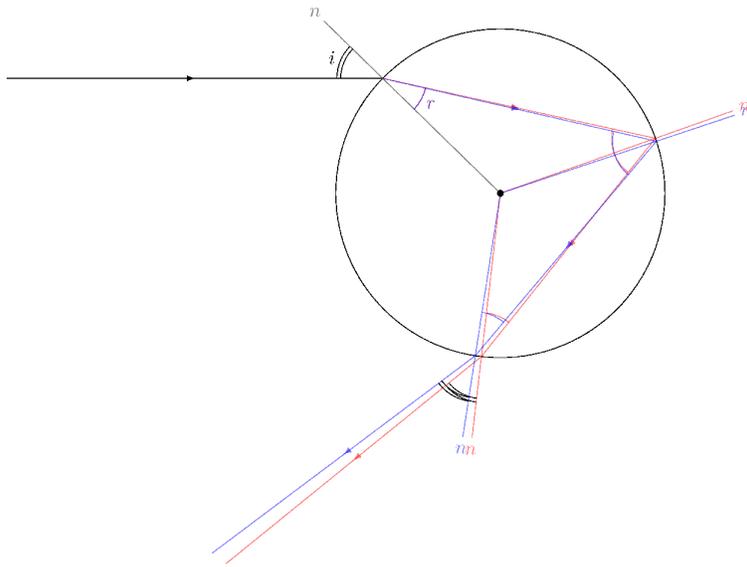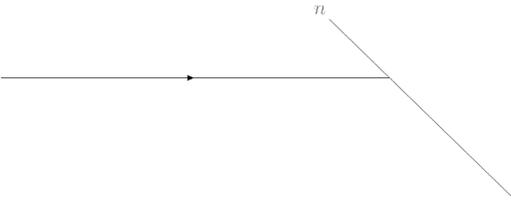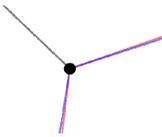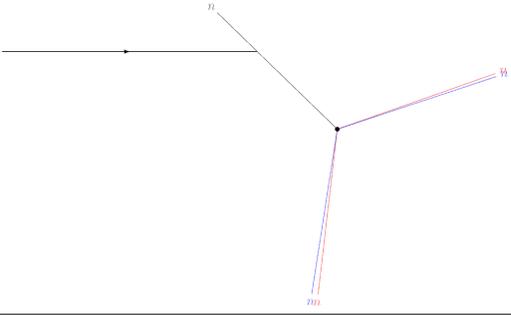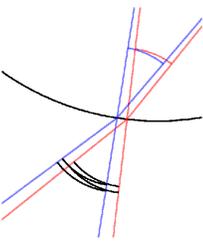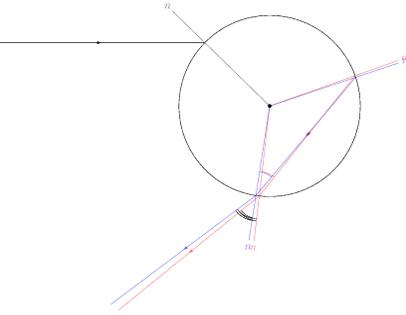
Fig. 3. A picture written using the TikZ/PGF drawing language and three of its slices. Unlike Figure 2, the slice (the source code) is not shown because it does not easily fit in the figure. The original source and that of the first slice are shown in Figure 4.

(highlighted in Figure 4 with "+" signs) is similar to the first in that it successfully removes much of the original image. The retention of the line with the arrow appears to have the same cause as the retained diagonal line from the first example.

The third slice has the most complex slicing criteria. Here again the slicer is very successful in removing unwanted elements of the image. As with the prior two slices, it retains a few elements that prevent the image from shifting.

In summary, for RQ_3 VORBS produces slices that, when rendered, make it visually apparent that the correct portion of the picture description source code has been extracted. In some cases the rendered image appears to include unnecessary elements. These have two primary causes: dependences such as those seen in the bottom slice of Figure 2 and the negative impact of subpixel interpolation errors as seen in the first slice of Figure 3. The impact of these issues is comparatively minor and, visually, the slice still represents a significant reduction.

D. Threats to Validity

This section concludes by considering threats to validity. The most obvious threat is the limited number of PDLs considered. Considering more than the two languages, TikZ/PGF and pic, would help with the external validity. As this is an initial exploratory study there are no threats to the statistical validity. Internal validity is a concern as the off-the-shelf normalized squared difference computation from OpenCV is not designed specifically for slicing and thus does not capture exactly the desired relation. Future work will consider replacements designed to specifically support slicing of PDLs.

V. RELATED WORK

There is no work directly related to the slicing of picture description languages. This section briefly considers work related to the ORBS' approach and then compares it with traditional graphic clipping, the closest related operation from image processing. The original ORBS algorithm is related to Dynamic Slicing [28], [40], Critical Slicing [32], and delta-debugging [41] based approaches such as STRIPE [42] or Delta [33].

While ORBS computes observation-based slices, it is similar in intent to dynamic slicing, which has been implemented in many research prototypes [43]–[46]. With one exception existing dynamic slicing algorithms all apply to a single specific programming language and, furthermore, involve complex program analysis. Recently, Pócza et al. presented a multi-language dynamic slicing approach for .NET [47]. The key to their approach is leveraging the Common Language Runtime (CLR) debugging framework to provide traceability between instructions and the source code of different languages.

In terms of the underlying technique, the work closest to observation-based slicing is Critical Slicing [32] where a statement is considered critical if its deletion results in a changed behavior for the slicing criterion. A critical slice consists of all the critical statements. One limitation of this approach is that it considers statements to be critical although they may not be, and thus could be deleted after another statement were deleted (e.g., deleting a variable declaration first requires the deletion of all its uses). Thus, critical slices

```
+\begin{tikzpicture}[xscale=-1,
+    ray/.style={decoration={markings,mark=at position .5
+        with { \arrow[>=latex]{>}}},postaction=decorate}
+    ]
+    \pgfmathsetlengthmacro{\r}{3cm}
+    \pgfmathsetmacro{\f}{.7}
     \pgfmathsetlengthmacro{\arcradius}{.8cm}
     \pgfmathsetlengthmacro{\dotradius}{.6cm}
     \pgfmathsetlengthmacro{\arclabelradius}{1cm}
+    \pgfmathsetmacro{\incidentangle}{asin(\f)}
+    \coordinate (O) at (0, 0);
+    \coordinate (A) at (\incidentangle:\r);
     \draw (O) circle (\r);
+    \draw[ray] (A) -| \r*3, 0) -- (A);
+    \draw[gray] (O) -- ($(O)!1.5!(A)$) node[pos=1.05] {$n$};
     \drawarcdelta{(A)}{0}{\incidentangle}{\arcradius-1pt}
     \drawlabeledarcdelta{(A)}{0}{\incidentangle}
         {\arcradius+1pt} {$i$}{\arclabelradius}
+    \foreach \index/\color in {1.32/red, 1.34/blue} {
+        \pgfmathsetmacro{\refractedangle}
+            {asin(sin(\incidentangle) / \index)}
+        \pgfmathsetmacro{\angleindrop}
+            {180 - 2*\refractedangle}
+        \coordinate (A') at (\incidentangle+\angleindrop:);
+        \coordinate (A'') at (\incidentangle+2*\angleindrop:\r);
+        \begin{scope}[opacity=.5, color=\color]
+            \draw[ray] (A) -- (A');
             \draw[ray] (A') -- (A'');
             \draw[ray] (A'') -- ($(A'')+(2*\incidentangle
                 +2*\angleindrop:2*\r)$);
+            \draw (O) -- ($(O)!1.5!(A')$) node[pos=1.05] {$n$};
+            \draw (O) -- ($(O)!1.5!(A'')$) node[pos=1.05] {$n$};
             \drawlabeledarcdelta{(A)}{\incidentangle+180}
                 {-\refractedangle}{\arcradius}{$r$}
                 {\arclabelradius}
             \drawarcdelta{(A')}{\incidentangle+\angleindrop
                 +180}{\refractedangle} {\arcradius}
             \drawarcdelta{(A')}{\incidentangle+\angleindrop
                 +180}{-\refractedangle} {\arcradius}
             \drawarcdelta{(A'')}{\incidentangle+2*\angleindrop
                 +180}{\refractedangle} {\arcradius}
+        \end{scope}
         \drawarcdelta{(A'')}{\incidentangle+2*\angleindrop}
             {\incidentangle} {\arcradius-1pt}
         \drawarcdelta{(A'')}{\incidentangle+2*\angleindrop}
             {\incidentangle} {\arcradius+1pt}
    }
+    \draw[fill] (O) circle (1.5pt);
+\end{tikzpicture}
```

Fig. 4. Raindrop and its slice Raindrop-3 denoted by lines beginning "+".

can be significantly larger than ORBS slices. They can also fail the semantic requirement of a slice as statements individually deletable may not be deletable collectively [12].

The idea to delete parts of a program to test input is most prominent in applications of delta debugging [42], [48]. Recently, Regehr et al. [49] exploit the syntax and semantics of C to produce four delta-debugging based algorithms to minimize C programs that trigger compiler bugs. One could integrate such an approach to observation-based slicing. However, this would sacrifice the language independence. Interestingly, Delta debugging has been applied to LaTeX documents that generate build errors [50]. A modification of this approach to use visual criteria might produce output similar to VORBS although likely less efficient [12].

Finally, while not a requirement of VORBS, the examples shown in the paper all use a rectangular image as the template $\mathcal{C}.\mathcal{T}$ in the slicing criteria. Under such a restriction a traditional graphic clipping algorithm could be used to clip the geometrical elements of the PDL file to create a reduced file similar to the slices produced here. This assumes the use of a vector graphics program such as xfig and not a bitmap image editor such as xpaint. For vector graphics programs, the elements of the PDL intersecting the clipping regions can be easily identified using the reverse rendering pipeline. Knowing the semantics of the PDL, these elements could be more precisely clipped than simple element deletion allows. For example, assuming the existence of an arc primitive, the clipped ellipse of the middle slice from Figure 2, could be replaced at the source level with an arc (that omits the bottom of the ellipse). However, slicing has at least three advantages over this approach. Foremost, the slice can be rendered as it includes all necessary supporting elements (e.g., those used for positioning). Second, slicing is not limited to geometrical boundaries. Any sub-shape or sub-diagram, of any geometrical extent or semantics, can handled using the slicing approach. Finally, slicing can be applied to a file with hierarchical shape structures or transform stacks, rather than simply to individual geometrical elements.

VI. Future Work

This paper considers the challenges in generalizing ORBS to work with languages that have non-traditional semantics, specifically PDLs. This initial investigation leverages several off-the-shelf techniques. Future work will consider techniques specifically designed for slicing of PDLs. A few such techniques are considered in this section.

To begin with, the computation of R is symmetric in the sense that for a given offset (values of x and y), T and I can be interchanged in the definition. This symmetry comes from the use of squared terms in both the numerator and the denominator of the function. One implication of this symmetry is that a slice can in theory omit part of the slicing criterion. In practice this happens in some of the slices. An example is shown in Figure 5. The "good" example, shown at the top, finds the rendered slice includes a *superset* of the criteria. While an exact match is preferred, slices often include additional components. The "bad" example, shown at the bottom, finds the rendered slice includes a *subset* of the criteria. The key point here is that from the perspective of the R computation these two are the same because the computation is symmetric in the values of I and T. Projecting Weiser's original idea of what a slice preserves into the generalization, one would expect that, at a minimum, everything in the template (and as little of the remaining picture as possible) would be included in the slice. Thus future work will consider replacing CV_TM_SQDIFF_NORMED in the computation of R with a slicing-specific computation.

As mentioned in Section IV, future work will also consider the impact of rasterization, whose effect is illustrated in Figure 6. The figure was generated by shifting the sub-image to be matched in increments of 0.001pt (there are 72 points (pt) to the inch). There is a clear cyclic pattern to the R values. Pragmatically, what this means that any large scale movement that otherwise leaves the image unchanged will land somewhere in this cycle. If R_0 happens to be close to the


Fig. 5. Impact of R's symmetry on the slice on hydrogen-3.
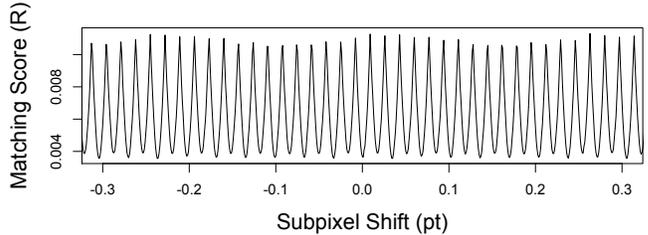
Fig. 6. Interaction between subpixel movement and rasterization

bottom then it is very unlikely that a large scale shift will be accepted because if is unlikely to produce a lower R value.

Some more obvious extensions include extending the set of languages considered to include, for example, Postscript, idraw, xfig, web pages (which are inherently multi-language including HTML, JavaScript, CSS), and GUI codes such as Java applets. The image matching for these applications will have to cope with additional noises, such as browser compatibilities for CSS and HTML. Another example extension is to consider the deletion of alternate lexical units. The current implementation works at the line-of-text level. One obvious alternative is the white-space-separated-token level. This change would allow, for example, the removal of the text "**program** *Main*", "ENTRY", and "*i := 1*" from the last slice shown in Figure 2.

VII. Conclusion

This work builds on ORBS, the first language-independent program slicer that computes slices for systems written in multiple languages. ORBS exploits statement *deletion* as its primary operation and *observation* as its validation criteria.

This paper describes the generalization of ORBS to languages with non-traditional semantics. As an initial case study it considered picture description languages. The resulting algorithm VORBS is very effective at slicing picture descriptions. The resulting slices can be used where only a subset of the original image is required.

The creation of VORBS illustrates the viability of generalizing slicing to languages with non-traditional semantics. VORBS observes and compares the behavior of PDLs using the normalized sum of square differences. VORBS was quite successful leading to an average 85% reduction in code size. Using VORBS as a model it should be possible to generalize the ORBS' approach to other languages with non-standard semantics.

REFERENCES

[1] M. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. dissertation, University of Michigan, Ann Arbor, MI, 1979.

[2] L. D. Larsen and M. J. Harrold, "Slicing object-oriented software," in *Proceedings of the 18th International Conference on Software Engineering*, Berlin, 1996, pp. 495–505.

[3] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *In Int. Conf. on Softw. Maint*, 1997, pp. 188–195.

[4] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li, "Model projection: simplifying models in response to restricting the environment," *Software Engineering, International Conference on*, pp. 291–300, 2011.

[5] J. Choi and J. Ferrante, "Static slicing in the presence of goto statements," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, pp. 1097–1113, Jul. 1994.

[6] M. Harman, A. Lakhotia, and D. W. Binkley, "A framework for static slicers of unstructured programs," *Information and Software Technology*, vol. 48, no. 7, pp. 549–565, 2006.

[7] M. Harman and S. Danicic, "Amorphous program slicing," in 5th *IEEE International Workshop on Program Comprehesion (IWPC'97)*. Los Alamitos, California, USA: IEEE Computer Society Press, May 1997.

[8] M. Harman, D. W. Binkley, and S. Danicic, "Amorphous program slicing," *Journal of Systems and Software*, vol. 68, no. 1, Oct. 2003.

[9] D. W. Binkley, "Precise executable interprocedural slices," *ACM Letters on Programming Languages and Systems*, vol. 3, no. 1-4, 1993.

[10] S. Horwitz, T. Reps, and D. W. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[11] J. Krinke, "Static slicing of threaded programs," in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, Jun. 1998, pp. 35–42.

[12] D. Binkley, N. Gold, M. Harman, J. Krinke, and S. Yoo, "Orbs: Language-independent program slicing," in *Foundations of Software Engineering*, Oct 2014.

[13] ——, "Observation-based slicing," University College London, Research Note RN/13/13, 2013.

[14] D. Binkley, "The application of program slicing to regression testing," *Information and Software Technology*, vol. 40, no. 11 and 12, 1998.

[15] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi, "Conditioned slicing supports partition testing," *Software Testing, Verification and Reliability*, vol. 12, pp. 23–28, Mar. 2002.

[16] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Software Engineering*, vol. 7, pp. 49–76, 2002.

[17] M. Weiser and J. Lyle, "Experiments on slicing-based debugging aids," in *Empirical Studies of Programmers: First Workshop*, 1985.

[18] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, Aug. 1991.

[19] Á. Hajnal and I. Forgács, "A demand-driven approach to slicing legacy COBOL systems," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 67–82, 2011.

[20] C. Cifuentes and A. Fraboulet, "Intraprocedural static slicing of binary executables," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 1997, pp. 188–195.

[21] J. Beck and D. Eichmann, "Program and interface slicing for reverse engineering," in *Proc. of the 15th International Conference on Software Engineering*, 1993, pp. 509–518.

[22] A. Cimitile, A. De Lucia, and M. Munro, "Identifying reusable functions using specification driven program slicing: a case study," in *Proc. of the International Conference on Software Maintenance (ICSM)*, 1995.

[23] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviours through program slicing," in 4th *International Workshop on Program Comprehension*, 1996, pp. 9–18.

[24] P. Tonella, "Using a concept lattice of decomposition slices for program understanding and impact analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 495–509, 2003.

[25] R. Ettinger and M. Verbaere, "Untangling: a slice extraction refactoring," in *Proc. of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, 2004, pp. 93–101.

[26] D. W. Binkley and K. B. Gallagher, "Program slicing," in *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1996.

[27] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.

[28] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[29] A. Podgurski and L. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, 1990.

[30] R. Parsons-Selke, "A graph semantics for program dependence graphs," in *Sixteenth ACM Symposium on Principles of Programming Languages (POPL)*, (Austin, TX, January 11-13, 1989), 1989, pp. 12–24.

[31] M. P. Ward, "Slicing the scam mug: A case study in semantic slicing," in *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, 2003, pp. 88–97.

[32] R. A. DeMillo, H. Pan, and E. H. Spafford, "Critical slicing for software fault localization," in *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, 1996, pp. 121–134.

[33] S. McPeak, D. S. Wilkerson, and S. Goldsmith, "Heuristically minimizes interesting files." delta.tigris.org.

[34] D. A. Wheeler, "SLOC count user's guide," www.dwheeler.com/sloccount/sloccount.html, 2004.

[35] S. Bhattacharjee and M. Kutter, "Compression tolerant image authentication," in *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, vol. 1, Oct 1998, pp. 435–439 vol.1.

[36] "SIPS - Scriptable Image Processing System: https://developer.apple.com/library/Mac/documentation/Darwin/Reference/ManPages/man1/sips.1.html."

[37] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[38] "PGF - a Portable Graphics Format for TeX: http://www.ctan.org/tex-archive/graphics/pgf/."

[39] B. W. Kernighan, "Pic - a language for typesetting graphics," *SIGPLAN Notes*, vol. 16, no. 6, pp. 92–98, April 1981.

[40] B. Korel and J. Laski, "Dynamic slicing in computer programs," *Journal of Systems and Software*, vol. 13, no. 3, pp. 187–195, 1990.

[41] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[42] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *International Workshop on Automated Debugging*, 2000.

[43] A. Beszedes, T. Gergely, Z. M. Szabó, J. Csirik, and T. Gyimothy, "Dynamic slicing method for maintenance of large C programs," in *Proc. of the 5th Conference on Software Maintenance and Reengineering*, 2001, pp. 105–113.

[44] G. Mund and R. Mall, "An efficient interprocedural dynamic slicing method," *Journal of Systems and Software*, vol. 79, no. 6, 2006.

[45] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, Apr. 2007.

[46] S. S. Barpanda and D. P. Mohapatra, "Dynamic slicing of distributed object-oriented programs," *IET software*, vol. 5, no. 5, 2011.

[47] K. Pócza, M. Biczó, and Z. Porkoláb, "Cross-language program slicing in the .NET framework," in *Proc. of the 3rd .NET Technologies Conference*, Plzen (Czech Republic), 2005, pp. 141–150.

[48] A. Zeller, "Yesterday, my program worked. today, it does not. Why?" in *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999.

[49] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Conference on Programming Language Design and Implementation*, 2012.

[50] O. A. Paraschenko, "Delta debugging for LATEX," uucode.com/blog/2011/04/27/delta-debugging-for-latex/.