



ELSEVIER

Contents lists available at ScienceDirect

## The Journal of Systems and Software

journal homepage: [www.elsevier.com/locate/jss](http://www.elsevier.com/locate/jss)

## Observational slicing based on visual semantics

Shin Yoo<sup>a</sup>, David Binkley<sup>b,\*</sup>, Roger Eastman<sup>b</sup><sup>a</sup>School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea<sup>b</sup>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 xxx

## 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.

© 2016 Elsevier Inc. All rights reserved.

## 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](mailto:shin.yoo@kaist.ac.kr) (S. Yoo), [binkley@cs.loyola.edu](mailto:binkley@cs.loyola.edu) (D. Binkley), [reastman@loyola.edu](mailto: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,  $\text{PROJ}_C$  (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  $i$ th 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 answer 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,  $Col$ . Slicing's deletion is restricted to the  $Col$ . This enables, for example, slicing programs that contain binary components and source code such as third-party libraries, which are excluded from  $Col$  and thus need not be changed by the slicer. Consequently, an observation-based slice, taken with respect to the criteria  $(v, l, \mathcal{I}, Col)$ , preserves the state trajectory for  $v$  at  $l$  for the selected inputs in  $\mathcal{I}$ , while deleting statements from the components of  $Col$  but no other components.

Furthermore, observation-based slicing is inherently 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  $C = (v, l, \mathcal{I}, Col)$  composed of variable  $v$ , line  $l$ , set of inputs  $\mathcal{I}$ , and components of interest  $Col$ , is any executable program with the following properties:

1.  $S$  can be obtained from  $P$  by deleting zero or more lines from  $Col$ .
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  $\text{PROJ}_C(T(P, I, v, l)) = \text{PROJ}_C(T(S, I, v, l))$ .

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

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 compiles but 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. ORBS systematically forms candidates until no more lines can be deleted.

Operationally, ORBS 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.

ORBS has successfully sliced programs known to be a challenge for traditional dependence-based slicers (Binkley et al., 2014; 2015) such as the one adorning the 2001 SCAM Mug (Ward, 2003). Furthermore, its slices compare favorably with those created by similar techniques such as Critical Slicing (DeMillo et al., 1996) and several slicing variants based on Delta Debugging (McPeak et al.). Finally, it successfully sliced the shell bash, which includes 118,167 source lines of code (SLOC) (Wheeler, 2004) and is written in eight different languages. The resulting slices include between 10% and 17% of the original program.

### 3. 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 on the ORBS algorithm 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 ORBS definition and the ORBS algorithm necessary to provide an effective slicing implementation for PDLs.

$RQ_2$ : *How much reduction is achieved by slicing?*

A key goal of slicing is to remove (slice away) unwanted parts of the source code.  $RQ_2$  takes a quantitative look at the effectiveness of the slicer at doing so for PDLs.

$RQ_3$ : *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, qualitatively, whether the slices appear visually correct. Thus the third research question considers the subjective visual correctness and precision of the new slicer.

### 4. Generalizing program slicing

Two challenges in slicing PDLs are generalizing the definition of a slice and extending ORBS accordingly. 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  $\mathcal{I}$  to preserving the visual appearance of template image  $T$  for inputs  $\mathcal{I}$ . For the

sake of the presentation simplicity, the rest of the paper (e.g., Algorithms 1 and 2) assumes that the picture descriptions have no input and that the set  $\mathcal{I}$  is a singleton set. Without the first of these, building an image (Lines 2 and 8 of Algorithm 2) would include  $C.\mathcal{I}$ . Without the latter assumption, an additional loop iterating over the inputs would be required.

#### Algorithm 1: ORBS (Binkley et al., 2014).

---

```

ORBSLICE( $P, C$ )
Input: Program  $P$ , slicing criterion  $C$ 
Output: A slice  $S$  of  $P$  for  $C$ 
(1)    $S \leftarrow \text{INSTRUMENT}(P, C)$ 
(2)    $T \leftarrow \text{EXECUTE}(\text{COMPILE}(S), C.\mathcal{I})$ 
(3)   ...
(4)   repeat
(5)     ...
(6)      $S' \leftarrow \text{next\_candidate\_slice}(S)$ 
(7)     if  $\text{COMPILE}(S') = \text{success}$ 
(8)        $T' \leftarrow \text{EXECUTE}(\text{COMPILE}(S'), C.\mathcal{I})$ 
(9)       if  $\text{PROJ}_C(T) = \text{PROJ}_C(T')$ 
(10)         $S \leftarrow S'$ 
(11)    ...
(12)   until no changes in  $S$ 
(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)    $R_0 \leftarrow \text{MATCH}(\text{COMPILE}(S), C.T)$ 
(3)   ...
(4)   repeat
(5)     ...
(6)      $S' \leftarrow \text{next\_candidate\_slice}(S)$ 
(7)     if  $\text{COMPILE}(S') = \text{success}$ 
(8)        $R \leftarrow \text{MATCH}(\text{COMPILE}(S'), C.T)$ 
(9)       if  $R \leq R_0$ 
(10)         $S \leftarrow S'$ 
(11)    ...
(12)   until no changes in  $S$ 
(13)   return  $S$ 

```

---

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 behavior of a candidate slice to that of 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* (Bhattacharjee and Kutter, 1998): 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 initial algorithm for slicing picture descriptions (Yoo et al., 2014) is a modification of the original ORBS algorithm for slicing traditional source code (Binkley et al., 2014). The core of the two 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<sup>1</sup>. The value five, used in the current implementation, was imperially derived as it provides a good balance between the computation cost of larger values and the increased slice size of smaller values. Given the performance of the current algorithm, there is little motivation to consider more expensive options. The candidate slice,  $S'$ , replaces  $S$  as the current slice on Line 10 if  $S'$  compiles and produces the correct projected trajectory.

#### 4.1. $RQ_1$ – impacts of the generalization

Turning to  $RQ_1$ , to help understand the impact on the original ORBS algorithm of generalizing to PDLs, Algorithm 1 is modified producing Algorithm 2, VORBS (*visual ORBS*), an algorithm for slicing PDLs. The modification replaces the computation and comparison of trajectories with the computation and comparison of *template-matching scores*. There are a range of template matching algorithms that compare a target template image  $T$  to an image  $I$ . For example, *normalized Sum of Square Differences* (SSD) is a well known and widely studied metric for image registration with general applicability. It is effectively a least squares minimization for its translation parameters  $(x, y)$  and as such is robust to a linear transformation  $T = gl + b$  between image intensity values that might be corrupted by Gaussian noise. It performs as well as optimal matched filter correlation in straightforward image registration. If the target and base images differ by more complex geometric and intensity transformations, such as scaling and skew in position and non-linear changes in intensity, other more complex image registration algorithms have been developed (for examples, see the surveys by Zitova and Flusser, 2003 and Moigne et al., 2011). The approach used in this paper employs the SSD metric as the most straightforward and appropriate.

SSD Template Matching's best-case match occurs when template  $T$  is cropped from  $I$ . Here in theory, the score, denoted  $R(x, y)$ , 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).

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. Based on the resulting algorithm, a tool was built and used to investigate  $RQ_2$  and  $RQ_3$ , which investigate the quantitative and qualitative aspects of the resulting slices.

## 5. Initial experiments

The objective of this section is to consider the performance of an initial implementation of Algorithm 2, and consider issues that could arise. To meet this objective, this section first describes the experimental setup used in the empirical investigation. It then describes the tools used in the implementation and the subject PDLs considered in the experiment. Finally it takes an initial look at  $RQ_2$  and  $RQ_3$ . While the quantitative look of  $RQ_2$  finds impressive results, the qualitative investigation undertaken as part of  $RQ_3$  identifies several shortcomings. These are investigated in

Section 6, which leads to an improved implementation. The performance of the improved implementation is then considered in Section 7 where  $RQ_2$  and  $RQ_3$  are revisited.

### 5.1. Experimental implementation and setup

The initial implementation was produced using off-the-shelf components to implement the image processing aspect of the VORBS algorithm. Key among these components is an efficient matching metric. The specific metric implementation used is the normalized squared difference, implemented in OpenCV as the option named `CV_TM_SQDIFF_NORMED`. This function, which is applied to rasterized bitmap images, matches template  $T$  in image  $I$  by computing the following score,  $R$ , for each possible location  $(x, y)$  in image  $I$ :

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

Unlike the high-level algorithm, the implementation must deal with real-world engineering issues. For example, image registration depends on the consistency of the image rendering pipeline used to render the original image, the template, and the slice (Bhattacharjee and Kutter, 1998). The use of `pdflatex` and `groff` to generate the vectorized diagrams and `sips` to generate bitmaps for the registration was initially believed to provide a very stable pipeline where the resulting images were very consistent at the pixel level. This consistency means that any efficient image registration algorithm would likely satisfy the requirement of estimating translation to one pixel, including absolute pixel difference.

Putting the pieces together, the initial VORBS implementation, illustrated in Fig. 1, is written in Python (version 2.7.5) and uses `sips` (Scriptable Image Processing System, version 10.4.4; `SIPS`) to rasterize vector graphic into the JPEG format. It also uses the OpenCV library (version 2.4.9) (Bradski, 2000) to perform the template matching, and finally, the cropping is performed directly on the 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 version, 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`.

To study the implementation requires picture descriptions. Two Picture Description Language families are used in the initial experiments: TikZ/PGF (PGF) and `pic/eqn/troff` (Kernighan, 1981). 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 drawing language with macros, branches, and loops. The source code of the images used in the experiments includes equations typeset using the `eqn` pre-processor and limited `troff` directives.

The initial investigation considers five slices of each of the five picture descriptions shown in Table 1. These include slices of 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 Fig. 2. The rendered `pic` image, shown later (at the top of Fig. 9), is taken from a paper describing how to perform interprocedural program slicing using the System Dependence Graph (Horwitz et al., 1990). Table 1 summarizes these five subjects, as well as one used in a later experiment in Section 7.

### 5.2. Quantitative evaluation

To investigate  $RQ_2$ , the quantitative reduction achieved by VORBS, five sub-images were cropped out of each of five rendered

<sup>1</sup> While other approaches are possible, the linear and consecutive deletion approach, called *moving deletion window*, is preferred for performance reasons. In particular, with  $2^n$  possibilities, trying all possible subsets of an  $n$  line program is intractable. See the original ORBS experiments for further detail (Binkley et al., 2014).

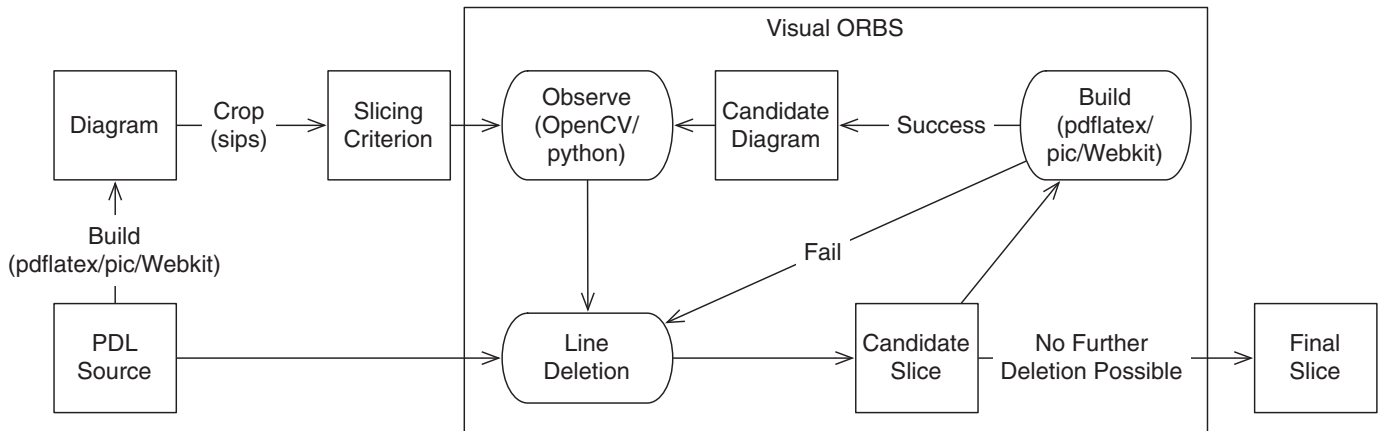


Fig. 1. The implementation of the VORBS algorithm.

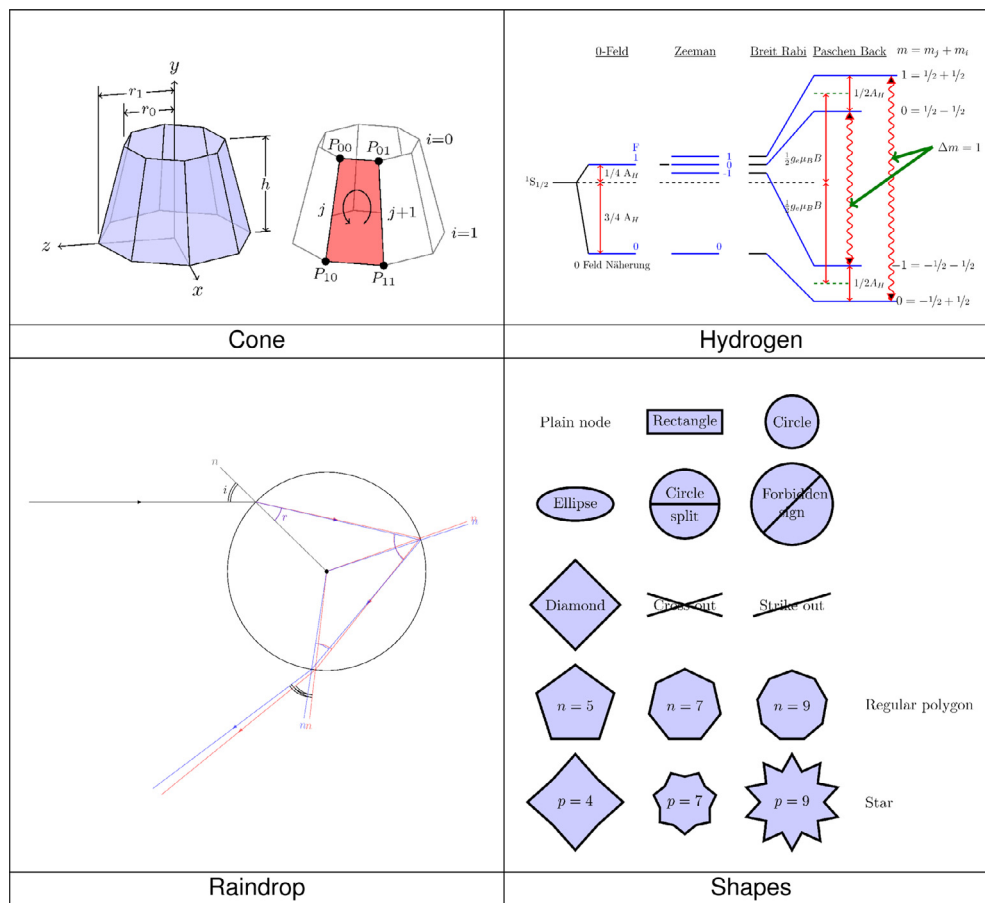


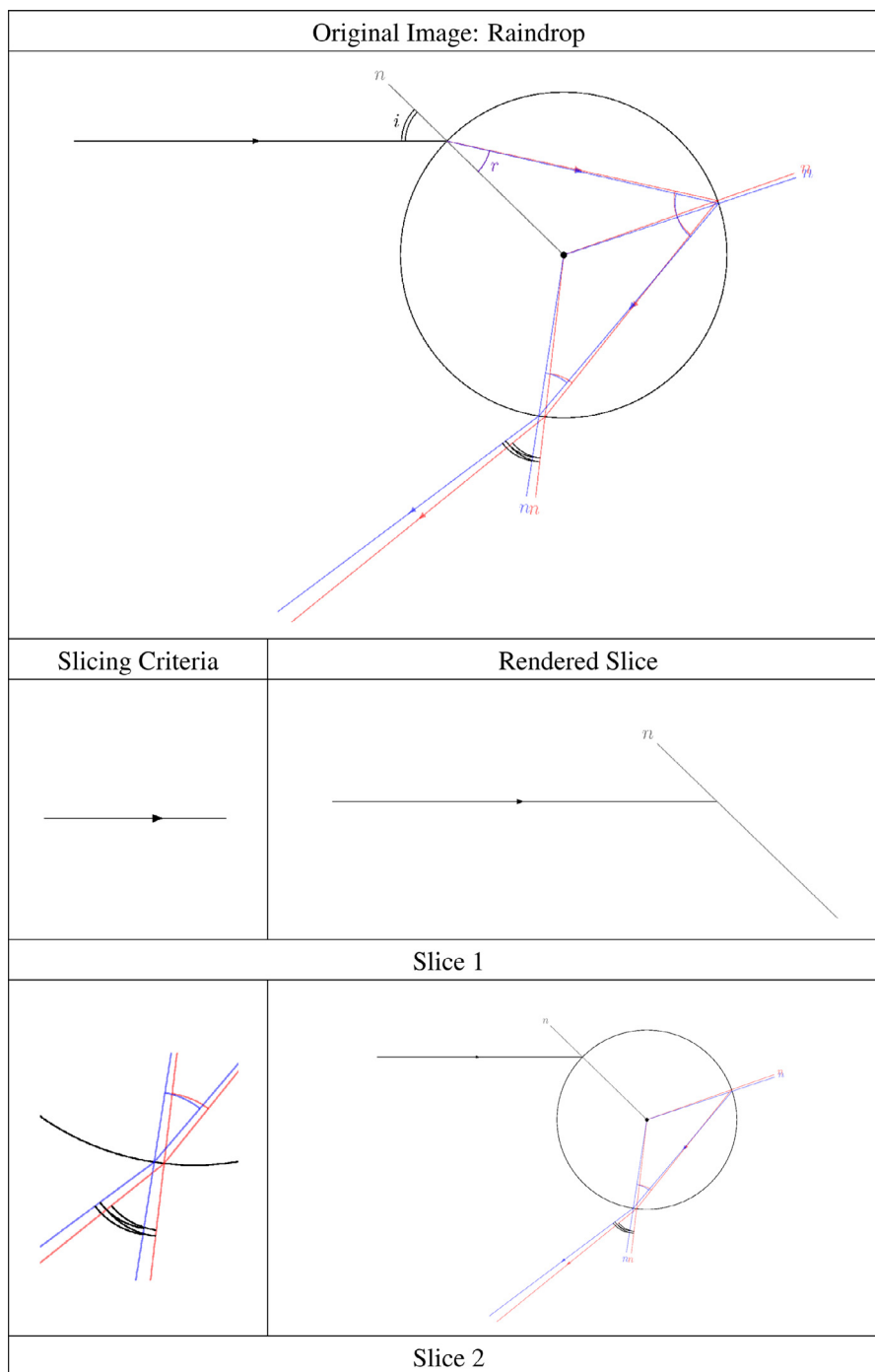
Fig. 2. The four TikZ/PGF images sliced.

Table 1 Studied picture descriptions.

Name	Language	LoC
Cone	TikZ/PGF	74
Hydrogen	TikZ/PGF	61
Raindrop	TikZ/PGF	44
Shapes	TikZ/PGF	25
PDG-figure	pic/eqn/troff	262
ThreeCols	HTML/CSS/Javascript	343

images and used as slicing criteria, producing a collection of 25 slices in total. These sub-images, some of which are shown in Figs. 3, 9, and 10 cover a range from simple to complex portions of the original images.<sup>2</sup> The 25 slices are summarized in Table 2 which shows the number of lines in the Col of the original image's

<sup>2</sup> 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://coinse.kaist.ac.kr/projects/visualorbs/>.



**Fig. 3.** A picture written using the TikZ/PGF drawing language and two of its slices. Only the rendered slice is shown because the slice (the source code) does not easily fit in the figure. The original source and that of Slice 2 are shown in Fig. 4.

source code, the number of passes the slicer used, the total number of lines deleted, and finally the percent size reduction in the picture description source code. The number of passes refers to the number of iterations of the slicer's outermost loop (Line 4 of Algorithms 1 and 2). 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 percent reduction, 85%, is substantial.

The percent reduction is given as a percentage of the Col. In the experiments the Col 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 Col 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' initial implementation efficiently and automatically extracts, from a picture description, the code that corresponds to a particular sub-image.

**Table 2**  
Initial picture descriptions slice statistics.

Subject	Lines in Col	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%

### 5.3. Qualitative evaluation

To counterbalance RQ<sub>2</sub>'s quantitative look, RQ<sub>3</sub> considers the slices qualitatively. This section focuses on slices that suggest shortcomings in the initial implementation. Three examples, all slices of the TikZ picture descriptions, are considered: the two Raindrop slices shown in Fig. 3 and the Hydrogen slice shown in Fig. 5. The PDL source for the Raindrop image is shown in Fig. 4 along with one of its slices.

Considering each example slice in turn, Slice 1 of Fig. 3, 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 additionally omit the diagonal line segment labeled *n*. In the source shown in Fig. 4, this line segment is the third `\draw` command, labeled ①; the line with the arrow does not depend on it. However, its removal shifts the rendered image up. At first glance it appears that this should not impact the computation of *R*, but it does. An investigation of this phenomena, presented in Section 6, suggests that subpixel interpolation errors introduce a kind of blurring that negatively effects *R*'s usefulness.

The second slice, Slice 2 of Fig. 3, uses a more complex slicing criterion. Here again the slicer is very successful in removing unwanted elements of the image. As with the prior slice, it retains three elements that prevent the image from shifting. These three include the diagonal line segment retained in the first slice, the leftmost line segment with the arrow head, and the rightmost refraction of the diagonal line segment. By maintaining the width and the distance from the top of the rendered image, retaining these three elements preserve the other elements location and thus prevent subpixel interpolation errors.

The third example, Hydrogen-3, illustrates a shortcoming of using a symmetric, off-the-shelf, matching algorithm. The slicing criteria is the energy states appearing in the center of the original image (the image Hydrogen shown in Fig. 2). The slicing criteria is shown in the lower left of Fig. 5. The core of the issue here is that 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 equation. One implication of this symmetry is that a slice can in theory omit part of the slicing criterion. In practice this occurs in a few of the slices. Fig. 5 shows an example where “The Good” example, shown at the top of the figure, 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 rendered slice. Thus there is a need to consider replacing CV\_TM\_SQDIFF\_NORMED in the computation of *R* with a slicing-specific computation.

In summary, for RQ<sub>3</sub> most of the slices produced by the initial implementation of VORBS, when rendered, make it visually apparent that the correct portion of the picture description source code has been extracted. However there is a need to deal with the subpixel interpolation error and to devise a slicing-specific matching function.

## 6. Challenges and enhanced implementation

This section investigates the causes of the challenges observed in Section 5, including subpixel interpolation errors and the need for asymmetry in the matching function. As a result of these investigations, several enhancement to the VORBS' implementation are made. The improved implementation is studied in Section 7.

Considering first the subpixel interpolation errors, it turns out that the use of JPEG compression for intermediate image storage can lead to the subpixel jitter effects in the matching function making the computed value for *R* inconsistent. JPEG compression (pre-JPEG2000) uses discrete cosine transformation (DCT) on small blocks of the image. It throws away high frequency, low energy coefficients. What remains are fewer, low bit count, coefficients.

This can lead to subpixel shifts. For example, consider moving a one pixel black vertical line from right to left by subpixel amounts. Assuming that initially the line is fully aligned with the image and is contained in one pixel column. Shift it slightly to the left and it blurs out to two pixel columns. That changes the DCT coefficients magnitude and phase. Keep shifting it and the blurring continues to happen, until it has moved exactly one pixel to the left. Then the line again aligns with a vertical pixel column. As an added challenge, this cyclic behavior also can occur on scales less than one pixel. The negative impact of subpixel shifts is illustrated in Fig. 6, which shows the imprecise rendered slice produced by the initial implementation and the desired rendered slice in the absence of sub-pixel interpolation artifacts.

An investigation of the impact of rasterization was performed by shifting a sub-image to be matched in increments of 0.001pt (there are 72 points (pt) to the inch). The resulting *R* values are graphed in Fig. 7 where harmonics on at least two frequencies are evident. This clear cyclic pattern to the *R* values means that any large scale movement that otherwise leave the image unchanged will land somewhere in the cycle. Pragmatically, if *R*<sub>0</sub> happens to be close to a minimum then it is very unlikely that a large scale shift will be accepted because it is unlikely to produce a lower *R* value.

This phenomena can happen even without JPEG compression – the subpixel shifts impose a blurring operator on the image which can alone change the matching (Pluim et al., 2000). Furthermore, since the figures being sliced are high contrast diagrams with large blank spaces and sharp dark features of lines and text,

```

-- \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] {$n$}; ①
--   \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:\r);
--     \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-4: lines included in slices are denoted with by “-” in the two leftmost columns. The first column marks lines in the slice produced by the initial implementation, while the second column marks lines the slice produced by the improved implementation studied in Section 7.

the edges of the features give rise to strong compression artifacts that can change with subpixel shifts. Such artifacts from subpixel movement and JPEG compression can be particularly addressed by moving from JPEG to TIFF or PNG, which are lossless image formats.

While working with the subpixel jitter, it was discovered that the rendering may also be inconsistent because the software implementing vector rasterization exhibited non-deterministic behavior. The resulting inconsistency can lead to poor matching results. In the case of the original implementation of VORBS, sips exhibited such non-determinism. In one particular case, when given a PDF that contained two separate and independent graphical ob-

jects, *A* and *B*, sips would rasterize object *A* differently when object *B* was deleted from the image. If the slicing criterion is object *A*, this can result in not being able to delete object *B*. To address this issue, the use of sips was replaced by the convert tool, which does not exhibit this phenomenon.

The second shortcoming of the initial experiments was caused by the symmetry in the off-the-shelf template matching function. VORBS needs a slicing-specific function that, at a minimum, preserves the template image in the slice. In visual terms any non-background from the template *must* exist in the slice. On the other hand, finding background where there is background in the template and everywhere outside the template is preferred.



Slicing Criteria	Rendered Slice
Example One – The Good	
Example Two – The Bad	

Fig. 5. Impact of  $R$ 's symmetry on the slice on Hydrogen-3.

The second challenge manifests itself in the need for an asymmetric matching function. A real-world illustration of this need is shown in Fig. 8 based on the example considered in Fig. 5. In this example the black line segment on the left is part of the slicing criteria and thus should be retained in the slice. However because of the symmetry in the matching function, it gets unwantedly omitted by the initial implementation resulting in an errant slice.

To elucidate the core issue a simplified conceptual example of the matching is presented in Table 3. This example shows a black image drawn on a white background. Slicing's goal is to "replace" as much of the black that occurs outside the slicing criteria with background. The original image includes a square and a circle. The goal is to slice its description with respect to the square. While any original image is (always a) slice of itself, it is often not a very good slice. In the example this is because it unwontedly includes the circle. The slice labeled "OK slice" is an improvement as it includes less non-background from outside the criteria (it includes only half the circle). Going from half to a quarter of the circle produces a better slice as the amount of non-background outside the criterion has been reduced. Finally, the last line is not a slice because non-background of the criterion (half of the square) is absent from the rendered image. In this final case it does not matter how much of the circle is present as the rendered image fails to include all of the square (i.e., all of the criterion).

A more rigorous explanation of the asymmetric matching is necessary to derive an appropriate matching function. Doing so involves digging down to the pixel level, as shown in Table 4. The explanation assumes that the background color is white and thus the slicer is attempting to remove non-white parts of the image except those that belong to the criteria (the template). The upper example is in black and white to simplify the initial explanation. The ideal case is when corresponding pixels in the template and the image are both white or both black, as is the case in the first and the fourth columns of the figure. In such cases the score,  $R$  in Algorithm 2, takes its minimal value of zero. For slicing, the remaining two cases need to be treated asymmetrically. First, as seen in the second column, when the slicing criteria contains black where the rendered candidate slice is white then the candidate needs to be rejected because it fails to include an element from the slice criteria (the template). In contrast, the case in the third column is an instance where the candidate slice includes something not found in the criteria. While this is not ideal, it does not mean that the candidate slice needs to be rejected and thus only serves to increase the  $R$  score.

The second example shown in Table 4 extends the first from black and white to gray scale using three grays: light gray (0xd3), gray (0xbe), and dark gray (0x4f). Because the template is white the darker the image the worse the score. Generalizing this obser-

vation, the greater the difference in the pixel value, the worse the match; thus the difference is added to the score. The current implementation reduces color images to gray scale, but future work will consider how an image with multiple colors should be treated.

To overcome the limitations of a symmetric matching function, a slicing-specific matching function was written based on the illustration given in Table 4. Its pseudo-code is shown in Algorithm 3. The algorithm treats as "background" all pixels with a value above 200 (images are converted to 8-bit grayscale while rendering). Setting this cutoff value to high (e.g., 250), did not work well. In the implementation if any pixel violates the condition "where the slicing criterion is non-background, the rendered slice must not be background" the potential slice is assigned a score of infinity. Otherwise the sum of differences at each pixel is used as the score. This process is repeated over the entire rendered potential slice (all values of  $x$  and  $y$ ). The minimum score and its location are returned.

Putting the pieces together, the improved VORBS implementation, is written in Python (version 2.7.9) and includes a template matching algorithm implemented in pure Python, using Pillow (version 2.6.1) for image file I/O and Numpy (version 1.9) for pixel matrix computation. All rasterized images are handled in lossless PNG format to avoid compression artifacts. The conversion from PDF to PNG is handled by the convert tool from ImageMagick (version 6.8.9-8). The slicing criteria images were also cropped using the convert tool. (The convert tool proved more robust when rasterizing PDF vector images than sips.) For TikZ/PGF source code, pdflatex (version 2.6-1.40.15) from TeXLive 2014 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.

## 7. Empirical investigation

Using the improved implementation this section describes a more comprehensive empirical study of VORBS. It repeats the 25 slices used in the initial experiments, considers several of these qualitatively, and then considers three slices of the web page "Perfect multi-column liquid layouts". This web page both uses and explains the use of a three column liquid layout, has been tested on multiple browsers, and is iPhone compatible.<sup>3</sup> The analysis again first takes a quantitative look at the slices and then a qualitative look and thus provides answers to first  $RQ_2$  and then  $RQ_3$ .

### 7.1. Quantitative evaluation

Table 5 shows the quantitative data obtained using the improved VORBS implementation. As in the initial experiments, the percent reduction is given as a percentage of the CoI, which excludes the 23 line  $\text{\LaTeX}$  harness and the 33 files used to build the paper from which the pic image description was taken. For ease of comparison, the final column of the table repeats the percent reduction from Table 2 obtained using the initial implementation.

Many of the individual slices are the same as both versions produce the minimal slice. In several cases (e.g., Cone-5) the slice includes more of the original source code because of the stricter requirements imposed by the slicing specific matching. Overall the need to preserve the criteria leads to a decrease in the average reduction for the 25 slices when compared to the average in the initial experiment, from 85% to 83% (shown in the row labeled Average-25). The overall average, Average-all, is a bit lower still at 81%, because the first of the ThreeCols slices necessarily includes almost half of the original HTML source code.

<sup>3</sup> <http://matthewjamestaylor.com/blog/perfect-3-column.htm>.

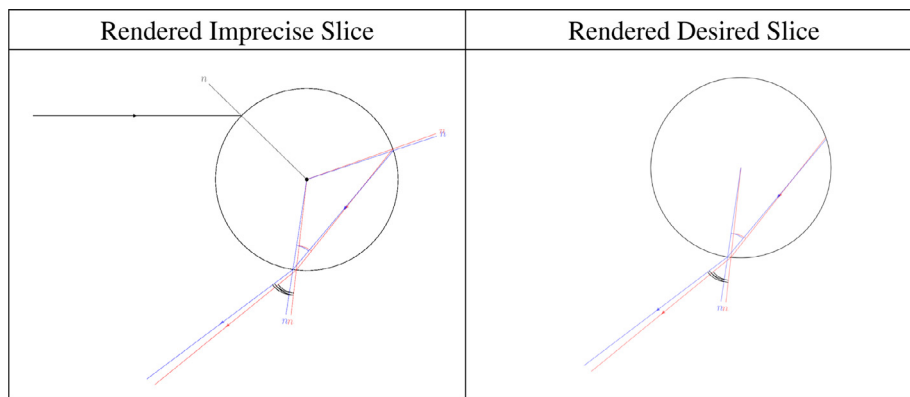


Fig. 6. Negative impact of subpixel jitter.

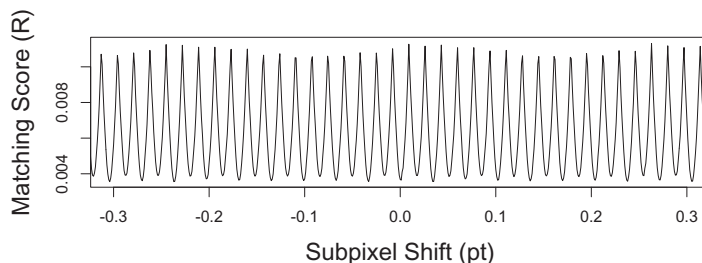


Fig. 7. Interaction between subpixel movement and rasterization.

Slicing Criterion	Imprecise Rendered Slice	Desired Rendered Slice

Fig. 8. Negative impact of symmetry in the template matching function.

Table 3  
Asymmetric matching example.

Original Image	
Slicing Criterion	
Rendered OK Slice	
Rendered Better Slice	
Rendered Non-Slice	

Table 4  
Asymmetric matching detail.

Black and White	Column	1	2	3	4
Template (slicing criterion)					
Image (candidate slice)					
Score		0	$\infty$	1	0
8-bit Grayscale	Column	1	2	3	4
Template (slicing criterion)					
Image (candidate slice)					
Score		0	0xd3	0xbe	0x4f

In summary, for RQ<sub>2</sub> the case study of 28 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. Thus, VORBS can automatically extract from a picture description the source code that corresponds to a particular sub-image.

### 7.2. Qualitative investigation

The qualitative look at the slices considers in detail three PDG-figure slices, a comparative look at the TikZ slices from Section 5,

**Algorithm 3** Slicing specific matching function.

```

SCORE(template, subimage, t)
(1)  sum ← 0
(2)  for i ← 0 to template.max_x
(3)    for j ← 0 to template.max_y
(4)      if template[i][j] < t and subimage[i][j] > t
(5)        return ∞
(6)      else
(7)        sum ← sum + template[i][j] - subimage[i][j]
(8)  return sum
MATCH(source, template)
(1)  t ← 200 - background "white" is any pixel whose value is "> t"
(2)  min_score ← ∞
(3)  for x ← 0 to source.max_x - template.max_x
(4)    for y ← 0 to source.max_y - template.max_y
(5)      s ← source[y : y+template.max_y, x : x+template.max_x]
(6)      if SCORE(template, s, t) < min_score
(7)        min_score = SCORE(template, s, t)
(8)        min_x = x
(9)        min_y = y
(10) return min_score, min_y, min_x

```

**Table 5**  
Picture descriptions slice statistics.

Subject	Lines in Col	Slicer passes	Deleted lines	Percent reduction	
				Improved	Initial
PDG-figure-1	262	5	251	96%	96%
PDG-figure-2	262	4	258	98%	98%
PDG-figure-3	262	3	249	95%	95%
PDG-figure-4	262	3	242	92%	92%
PDG-figure-5	262	3	245	94%	94%
Cone-1	74	4	35	47%	46%
Cone-2	74	3	36	49%	50%
Cone-3	74	3	68	92%	96%
Cone-4	74	3	68	92%	92%
Cone-5	74	3	60	81%	84%
Hydrogen-1	61	3	47	77%	82%
Hydrogen-2	61	3	53	87%	89%
Hydrogen-3	61	3	52	85%	89%
Hydrogen-4	61	5	51	84%	84%
Hydrogen-5	61	3	56	92%	93%
Raindrop-1	45	4	28	62%	62%
Raindrop-2	45	4	35	78%	73%
Raindrop-3	45	3	25	56%	49%
Raindrop-4	45	4	17	38%	27%
Raindrop-5	45	4	39	87%	87%
Shapes-1	25	3	4	16%	36%
Shapes-2	25	2	2	8%	44%
Shapes-3	25	3	4	16%	68%
Shapes-4	25	2	4	16%	28%
Shapes-5	25	3	15	60%	56%
ThreeCols-1	343	3	198	58%	na
ThreeCols-2	343	5	307	90%	na
ThreeCols-3	343	3	283	83%	na
Average-25		3.3		83%	85%
Average-all		3.3		81%	na

one other TikZ slice, and finally the result from applying VORBS to a web page. To begin with Fig. 9 shows the rendered example pic image, which was taken from a paper on how to perform program slicing using the System Dependence Graph (Horwitz et al., 1990). 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, labeled Slice 1, Slice 2, and Slice 3, are shown (these are slices PDG-figure-2 – PDG-

figure-4 from Table 5). 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 1, 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, Slice 2, 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 (labeled ④ in Fig. 9), which is shown wrapped around in the figure (wrapped lines begin with a + sign). This line draws the ellipse from which the slicing criterion was clipped. Its 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 (labeled ① Fig. 9) seems unrelated to the pic code. The connection comes from the inclusion of "@gsiz 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 rendered 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 also required for a reason similar to that of the "gsiz"

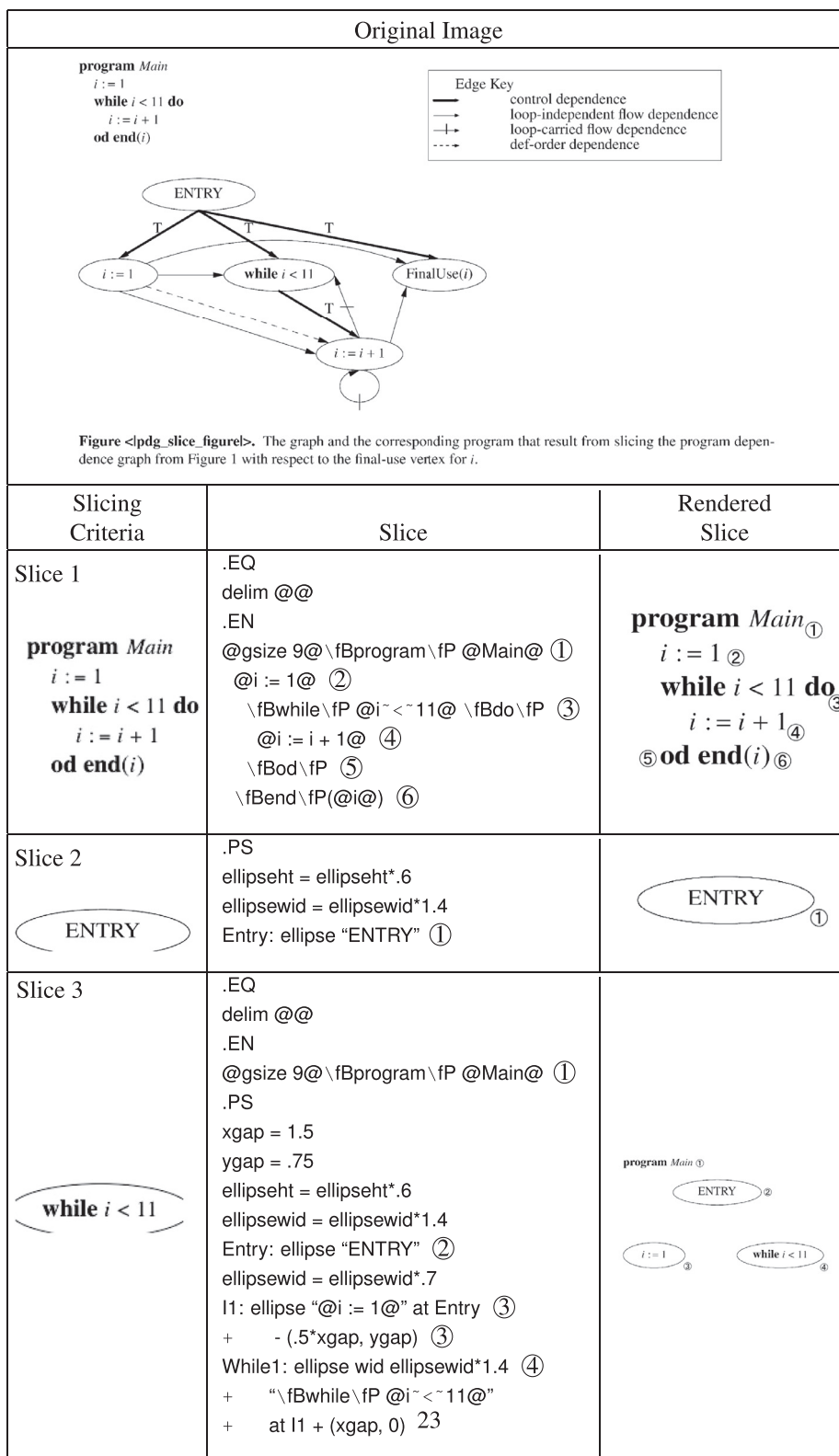
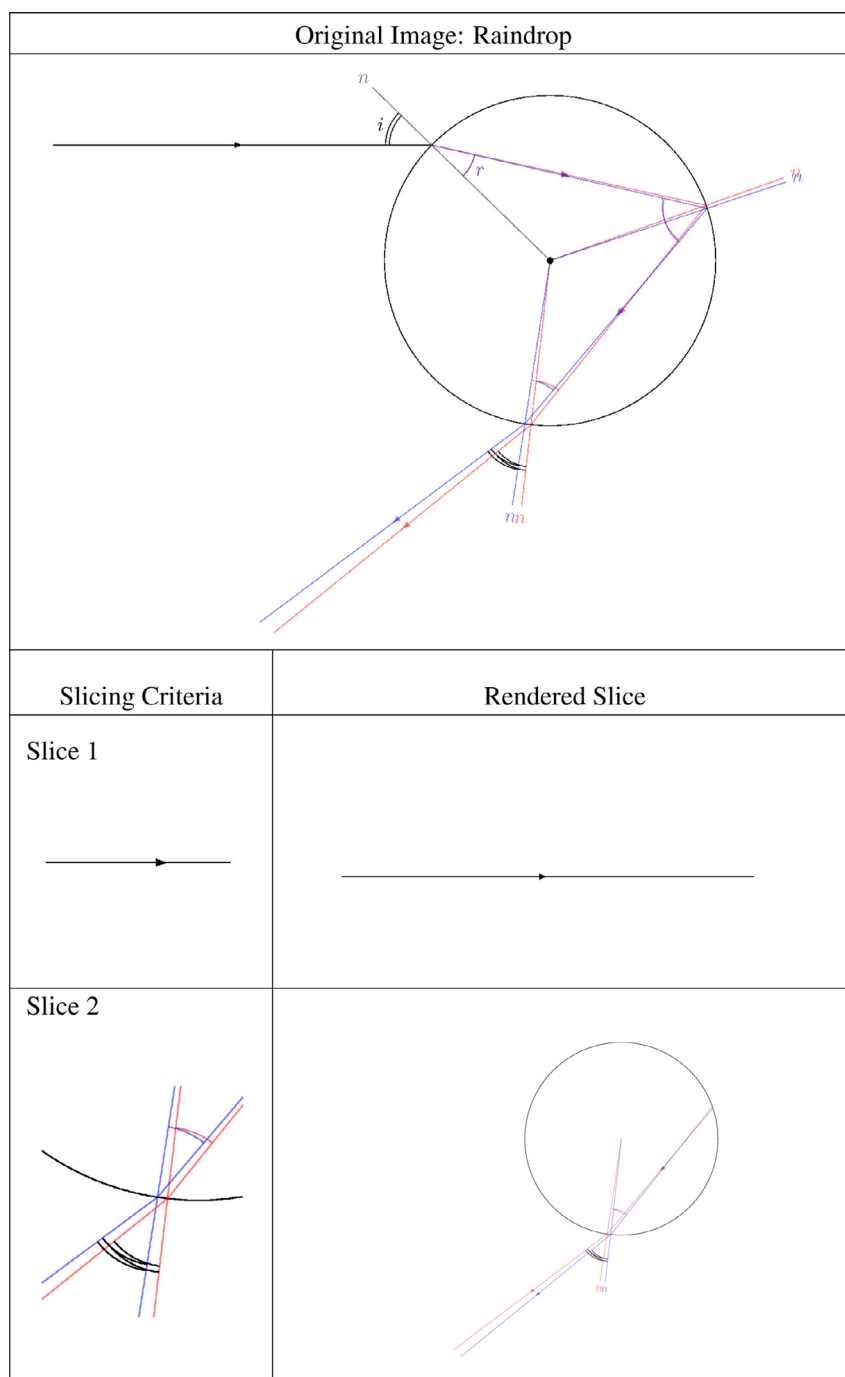


Fig. 9. Three slices of a picture written using pic and eqn.

directive. Inclusion of these four lines is an excellent example of the power brought by the use of observation. The VORBS slicer is able to capture dependencies 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 positions determine the position of the while vertex. The pic description of an element has the following syntax, <name>: <entity> "<label>"<location>



**Fig. 10.** Two rendered slices of Raindrop produced by the new implementation. The original source and that of the second slice are shown in Fig. 4.

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 (labeled ③ in Fig. 9). Transitively, this then requires the inclusion of ygap and the element defining the name “Entry” (labeled ②).

As an aside, capturing semantics using “pixel comparison” is a rather machine friendly approach. In this example, a human might prefer the smaller slice obtained by removing the “@gsize@” line even though when rendered it uses a larger font size. Capturing such perception might prove an interesting direction for future work.

Moving on to the next two slices, Fig. 10 revisits the two images from Raindrop considered in Fig. 3. The slices of this image

(Raindrop-2 and Raindrop-4 from Table 5) illustrate the language independence of VORBS as the same slicer constructed these slices and those in Fig. 9. Considering the two slices in turn, Slice 1 was taken with respect to the line with the arrow in the upper left of the original figure. Unlike the initial algorithm, which retained the diagonal line segment labeled  $n$ , to avoid the jitter caused by movement of elements of the rendered image, the improved implementation of VORBS correctly omits this unwanted line segment. (In the source shown in Fig. 4, this line segment is drawn by the third `\draw` command, labeled ①.) The result is the minimal slice. Similarly, for Slice 2 VORBS successfully removes all the unnecessary elements of the image source code. The slice is again minimal.




Slicing Criteria	Rendered Errant Slice	Rendered Correct Slice
		

Fig. 11. For the TikZ figure Hydrogen-3, the initial errant slice and the correct slice obtained using the asymmetric matching function.

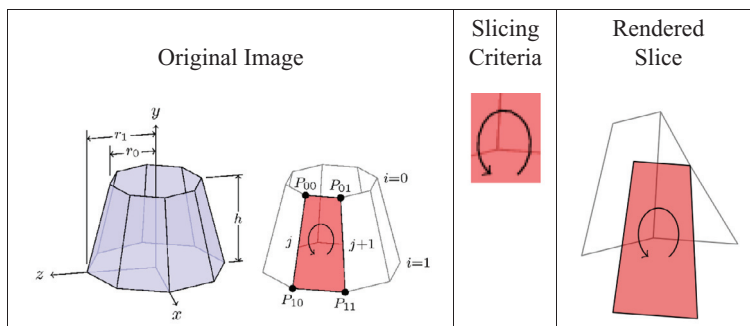


Fig. 12. Slice Cone-5.

In addition to revisiting the two Raindrop slices, the TikZ slice Hydrogen-3, shown in Fig. 11, is also reconsidered. This slice motivated the need for an asymmetric matching function. The figure includes the slicing criterion, the rendered version of the errant slice produced by the initial implementation, and the rendered version of the correct slice produced by the new implementation. As with several other TikZ slices, the slice is minimal, including exactly those elements needed to produce the criteria.

The final TikZ slice considered, Cone-5, shown in Fig. 12, helps illustrate why the scoring function does not consider the area outside the template image. The criteria includes a pink (lightgray) face and an arc from the front of the cone, and also the intersection of three line segments from the back of the cone. These line segments are part of the faces at the back of the cone.

In the rendered slice these elements are all present, which is good. However to include the intersection from the back of the cone, VORBS must include the two surfaces that have lines incident on the intersection. One of these faces includes the corner found at the far left of the rendered slice. This corner is obscured in the original image; thus from a scoring point of view this might appear as an unwanted non-background addition and thus cause to increase the score. Such a scoring function, which penalized a candidate slice because it included non-background elements outside the slicing criteria, might inhibit the slicer's ability to create this minimal slice. The current implementation ignores the image outside the template area. It relies on the deletion of source lines of image source code to minimize the amount of such rendered material.

To increase the external validity of VORBS language independence, the improved implementation was applied, unmodified, to a rendered web page. Modern web pages are inherently multi-language, often including (X)HTML(5), JavaScript, CSS, as well as other languages. The web page chosen "Perfect multi-column liquid layouts" describes (using itself) the use of a three column liquid layout that has been tested on multiple browsers. Its rendered form is shown in Fig. 13.

Fig. 14 shows three slicing criteria and the resulting rendered slices. The rationale behind the first slicing criterion, Slice 1 in the figure, is an attempt to preserve the menu and the three column layout but slice away unnecessary HTML and CSS elements. When applied to this example VORBS successfully removes unne-

cessary inline JavaScript as well as inline CSS. It could just as easily have removed references to external files or lines from those files. Slice 2 successfully extracts the minimal CSS and HTML to render the two labels found in the slicing criteria. Finally, the expectation of the final slice is the same as that of Slice 2, but as can be seen in the rendered version it includes unwanted elements from the web page. This shortcoming is similar to that found with the slices of Shapes and is considered in Section 9 with other future work.

A final comment on the web-page slices is that, like the pic slices, most tools that render HTML employ permissive parsing (e.g., accepting "errant" inputs such as missing closing tags). In this case all the slices omit closing `</script>`, `</body>`, and `</html>` tags. Finally, the advantages of observation are again underscored by these examples as dependence analysis of HTML and JavaScript, given their loose semantics, is a challenge for semantics-based tools such as slicers that need formal dependence models.

In summary, for RQ<sub>3</sub> 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: dependencies such as those seen in Slice 3 from Fig. 9 and the unwanted impact of subpixel interpolation errors as seen in Slice 3 of Fig. 14 and discussed further in Section 9. Despite these issues, visually, the rendered slices show a significant reduction.

### 7.3. 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 three families of languages, pic/eqn/troff, TikZ/PGF and HTML/CSS/JavaScript, would help with the external validity. Several other standard threats apply, such as the correct functioning of the tools used. Internal validity is a concern primarily because of the non-determinism found in the rendering pipeline. This issue, which makes the design of a slicing specific matching function a challenge, is considered further in Section 9. Future work will consider improvements to the matching aimed at better support slicing of PDLs. Along similar lines, the use of only five example slices might not be sufficient for

[Back to the CSS article by Matthew James Taylor](#)

## The Perfect 3 Column Liquid Layout (Percentage widths)

No CSS hacks. SEO friendly. No Images. No JavaScript. Cross-browser & iPhone compatible.

3 Column Holy Grail | 3 Column Blog Style | 2 Column Left Menu | 2 Column Right Menu | 2 Column Double Page | 1 Column Full Page | Stacked columns

Measure columns in: Pixel widths | Em widths | Percentage widths

**No CSS hacks**  
The CSS used for this layout is 100% valid and hack free. To overcome Internet Explorer's broken box model, no horizontal padding or margins are used in conjunction with a width. Instead, this design uses percentage widths and clever relative positioning.

**SEO friendly 2-1-3 column ordering**  
The higher up content is in your page code, the more important it is considered by search engine algorithms (see my article on [link source ordering](#) for more details on how this affects links). To make your website as optimised as possible, your main page content must come before the side columns. This layout does exactly that: The center page comes first, then the left column and finally the right column (see the nested div structure diagram for more info). The columns can also be configured to any other order if required.

**Full length column background colours**  
In this layout the background colours of each column will always stretch to the length of the longest column. This feature was traditionally only available with table based layouts, but now with a little CSS trickery we can do exactly the same with divs. Say goodbye to annoying short columns! You can read my article on [equal height columns](#) if you want to see how this is done.

**No Images**  
This layout requires no images. Many CSS website designs need images to colour in the column backgrounds but that is not necessary with this design. Why waste bandwidth and precious HTTP requests when you can do everything in pure CSS and XHTML?

**No JavaScript**  
JavaScript is not required. Some website layouts rely on JavaScript hacks to resize divs and force elements into place but you won't see any of that nonsense here. The JavaScript at the bottom of this page is just my Google Analytics tracking code, you can remove this when you use the layout.

**Resizable text compatible**  
This layout is fully compatible with resizable text. Resizable text is important for web accessibility. People who are vision impaired can make the text larger so it's easier for them to read. It is becoming increasingly more important to make your website resizable text compatible because people are expecting higher levels of web accessibility. Apple have made resizing the text on a website simple with the pinch gesture on their multi-touch trackpad. Is your website text-resizing compatible?

**No Quirks Mode**  
This liquid layout does not require the XML declaration for it to display correctly in older versions of Internet Explorer. This version works without it and is thus never in quirks mode.

**No IE Conditional Comments**  
Only one stylesheet is used with this layout. This means that IE conditional comments are not needed to set extra CSS rules for older versions of Internet Explorer.

[Download this layout](#) (25kb zip file).

**Percentage dimensions of the holy grail layout**

All the dimensions are in percentage widths so the layout adjusts to any screen resolution. Vertical dimensions are not set so they stretch to the height of the content.

**Maximum column content widths**  
To prevent wide content (like long URLs) from destroying the layout (long content can make the page scroll horizontally) the column content divs are set to overflow:hidden. This chops off any content that is wider than the div. Because of this, it's important to know the maximum widths allowable at common screen resolutions. For example, if you choose 800 x 600 pixels as your minimum compatible resolution what is the widest image that can be safely added to each column before it gets chopped off? Here are the figures:

800 x 600  
Left & right columns: 162 pixels  
Center page: 357 pixels

1024 x 768  
Left & right columns: 210 pixels  
Center page: 459 pixels

**The nested div structure**  
I've colour coded each div so it's easy to see:

The header, colmid and footer divs are 100% wide and stacked vertically one after the other. Colmid is inside colmid and colleft is inside colleft. The three column content divs (col1, col2 & col3) are inside colleft. Notice that the main content column (col1) comes before the other columns.

**Browser Compatibility**  
This 3 column liquid Layout has been tested on the following browsers:

**iPhone & iPod Touch**

- Safari

**Mac**

- Safari
- Firefox
- Opera 9.25
- Netscape 9.0.5 & 7.1

**Windows**

- Firefox 1.5, 2 & 3
- Safari
- Opera 8.1 & 9
- Google Chrome
- Explorer 5.5, 6 & 7
- Netscape 8

**Valid XHTML strict markup**  
The HTML in this layout validates as XHTML 1.0 strict.

**This layout is FREE for anyone to use**  
That's right, you don't have to pay anything. If you are feeling generous however, link back to this page so other people can find and use this layout too.

[Download this layout](#) (25kb zip file).

**Centered menu compatible**  
This layout is fully compatible with my [cross-browser-compatible centered menus](#).

This page uses the Perfect 'Holy Grail' 3 Column Liquid Layout by Matthew James Taylor. View more [website layouts](#) and [web design articles](#).

Fig. 13. HTML example.

verbose or feature rich PDLs. Two other potential internal threats mentioned in the preceding discussion are the permissive parsing used in processing pic images and web pages, and the subpixel interpolation errors, which are considered further in Section 9. As this is an initial exploratory study of the area there are no threats to the statistical validity.

Finally, the gap between humans' visual perception of a "good enough slice" and the one that can be captured mechanically may limit the approach. For example, in the pic example, the line including "@gsize 9@" has to be retained to obtain a sufficient high score. However, omitting the line simply leads to using the default size of 10, which to a human might be preferable to the inclusion of the "@gsize 9@" directive.

## 8. Related work

There is no work directly related to the slicing of Picture Description Languages. This section first considers a related approach in dependence analysis and then briefly considers work related to ORBS. The original ORBS algorithm is related to Dynamic Slicing (Korel and Laski, 1988; 1990), Critical Slicing (DeMillo et al., 1996), and delta debugging (Zeller and Hildebrandt, 2002) based approaches such as STRIPE (Cleve and Zeller, 2000) or Delta (McPeak et al.).

Traditional program slicing algorithms work by identifying dependencies in a program and then performing some kind of closure over the dependencies to extract a slice (Horwitz et al., 1990). The conservative nature of this dependence analysis leads to slices being over approximations. While practically impossible, extracting semantic dependencies (Podgurski and Clarke, 1990) would avoid this over approximation. Such an approach was recently hinted at in the work of Jiang et al. who consider using a (V)ORBS like approach to identify semantic dependencies (Jiang et al., 2014).

While ORBS computes observation-based slices, it is similar in intent to dynamic slicing, which has been implemented in many research prototypes (Beszedes et al., 2001; Mund and Mall, 2006; Zhang et al., 2007; Barpanda and Mohapatra, 2011). With one exception existing dynamic slicing algorithms all apply to a single specific programming language and, furthermore, involve complex program analysis. Recently, Póczya et al. presented a multi-language dynamic slicing approach for .NET (Póczya et al., 2005). 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 (DeMillo et al., 1996) 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 was deleted (e.g., deleting a variable declaration first requires the deletion of all its uses). Thus, critical slices 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 (Binkley et al., 2014).

The idea to delete parts of a program to test input is most prominent in applications of delta debugging (Zeller, 1999; Cleve and Zeller, 2000). Recently, Regehr et al. (2012) 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 L<sup>A</sup>T<sub>E</sub>X documents that generate build errors (Paraschenko, 2011). A modification of

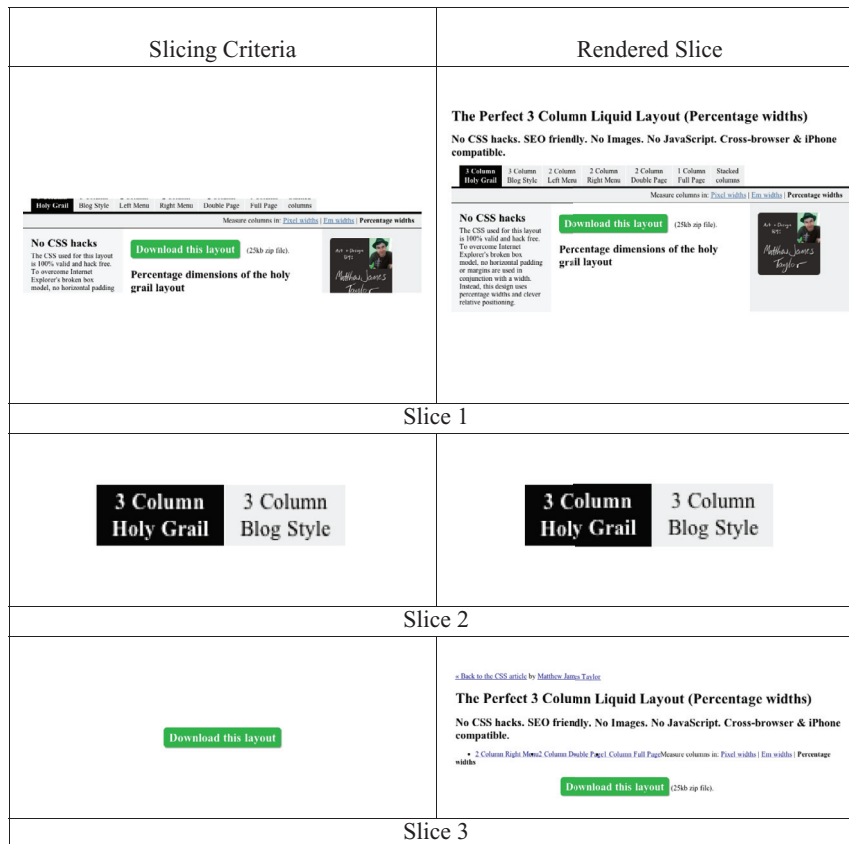


Fig. 14. Three rendered slices of the ThreeCols example.

this approach to use visual criteria might produce output similar to VORBS although likely less efficiently (Binkley et al., 2014).

## 9. Future work

Looking forward, the most significant challenge faced by VORBS is that for several reasons rendering tools are not required to be pixel-wise exact. Given an infinite number of pixels (dots) per inch, VORBS would not face this problem. In the real world the rendered images often must be split across pixel boundaries and are thus not pixel exact. Improvements to VORBS will address this inevitable subpixel interpolation.

Fig. 15 illustrates the problem using part of a Shape slice. The criteria for the slice is the circle cropped from the right of the original image, shown in the first row. In its first step, VORBS removes the source that renders to the label “Plain node”. As can be seen in the second row, this removal should be accepted because the rendered candidate slice clearly includes the circle. Unfortunately, when rendered, the circle differs between the criteria image and the sliced image at a subpixel level. Thus, at the step where VORBS should discover the presence of the criteria in the candidate slice, it fails. The reason why can be seen in a minimal energy image difference between the two (shown in the final row of the figure). In this difference image, black denotes a pixel with the same value in the two, white indicates that the first image includes background where the second does not, and finally gray indicates that the second image includes background where the first does not. Considered in the light of Table 4, it is a violation of the asymmetric matching constraint for the candidate slice to include background where the criteria does not, so the match fails.

There are several causes for this imprecision and in the extreme some per pixel differences are unavoidable. For example, Postscript uses single precision floats which can provide as few as six digits effective precision. Rounding errors are very likely. Furthermore, graphics specifications themselves allow non-determinism to accommodate some differences in implementation, such as by different graphics chips. Font rendering in browsers is particularly inconsistent.

In light of this example and the inevitability of pixel differences, we need to consider improved matching algorithms for VORBS. Clearly, doing correlations with thin lines, or images with high contrast elements, is sensitive to pixel shifts. Fortunately, there are subpixel algorithms that use techniques such as gradient descent to find small displacements. Thus VORBS could employ a matching function that is more robust and can handle small differences in pixel placement and values. A place for initial focus is if violations are related to the length of the boundaries in the target image. For example, assuming the target image includes the letter “O”, the length in pixels of the outer and inner edge gives an approximate count on the number of pixels that might be in violation. Another approach to increasing the robustness is to “blur” the image slightly (technically a morphological image dilation with a simple kernel), subtract the original from the new, and count the non-zero pixels. A third approach would employ machine learning. Many existing image algorithms need to have parameters, particularly thresholds, fine-tuned and machine learning approaches are often successful at doing so.

The current and near future work assumes that the original image and target do not differ significantly in intensity or color, since drastic color changes can cause drastic figure changes such as a foreground disappearing into a background. Future work will



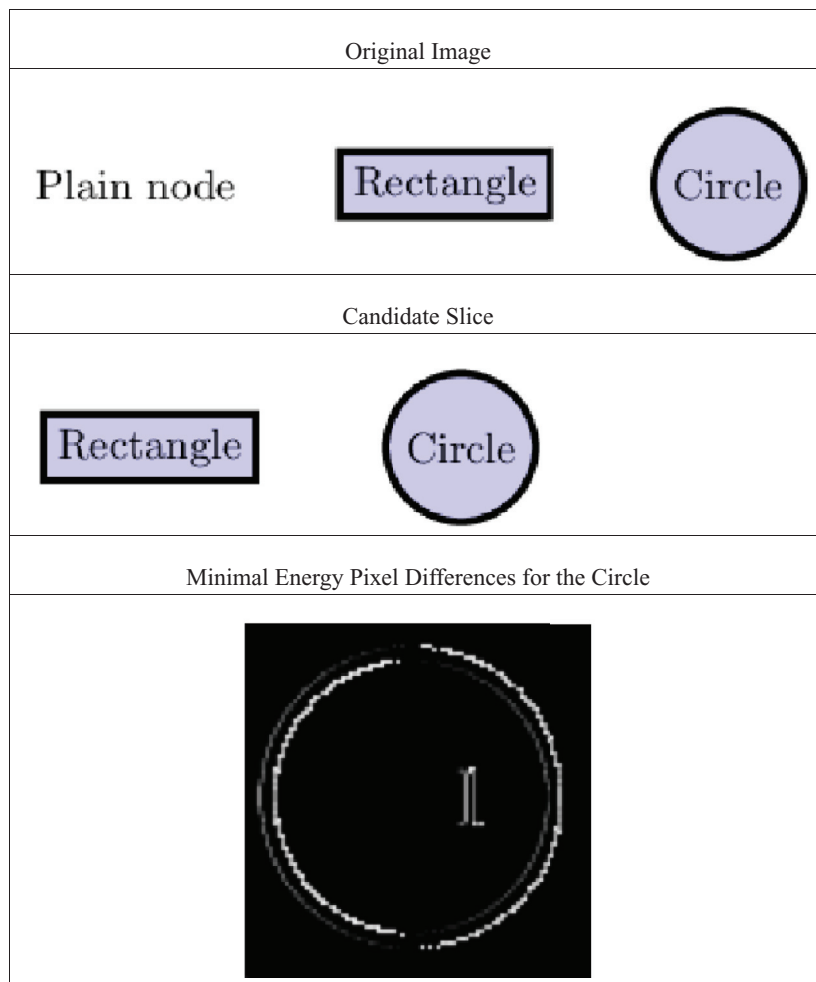


Fig. 15. Registration artifacts caused by subpixel motion.

need to consider how to capture the impact of color and intensity changes. The current algorithm can address the simple deletion of color commands using matching algorithms that work with systematic color changes.

Another future challenge to consider comes from the use of the matrix layout of the shapes in an image. The PDL may have a layout manager that can adjust relative positions of shapes. Once that is allowed template matching is more likely to fail. Alternatives include constraint-modified matching, which allows some inter-shape motion in matching. Such an algorithm basically recreates, in the matching function, the constraints allowed when the system lays out the diagram, which would make VORBS dependent on language-specific semantics.

Other extensions include growing the set of languages considered to include, for example, Postscript, idraw, xfig, and GUI codes such as Java applets. Another example extension would 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 Slice 3 of Fig. 9.

A more speculative direction for future work observes that, in comparison with traditional slicing, rather than picking a program component and asking the slicer to remove code while preserving the behavior of the selected component, VORBS takes a subset of the output and asks the slicer to slice away parts of the code that do not create the selected output. This change replaces the

variable,  $v$ , and line,  $l$ , of the slicing criteria with a subset of the output of the original program. The notion of providing as a slicing criteria a portion of the output is more general than requesting the preservation of a particular computation. (It is always possible to select those parts of the output that are produced by a particular computation. However, it is also possible to select only a subset of this output or portions output by several separate statements, something that cannot be captured using  $v$  and  $l$  alone.) This notion might be extended to an approach for automatic programming where a genetic algorithm proposes code and a slice extracts the subset of this code with a desired behavior.

Finally, while not a requirement of VORBS, the examples shown in the paper all use a rectangular, cropped image as the template  $CT$  in the slicing criteria. Under such a restriction a traditional graphic clipping algorithm with reverse picking 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 identified using the reverse rendering pipeline. Knowing the semantics of the PDL (something that VORBS ignores deliberately), 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 Slice 2 from Fig. 9, 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 clipping alternative.

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 be 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.

## 10. Conclusion

This work builds on ORBS, the first language-independent program slicer that computes slices for systems written in multiple languages. ORBS exploits line-of-text *deletion* as its primary operation and *observation* as its validation criteria. This paper describes the generalization of ORBS to languages with non-traditional visual semantics. As an initial case study it considered Picture Description Languages (PDL). The motivation for choosing PDL as our target includes providing a debugging aid for visual and representational faults, as well as a program comprehension tool for PDL users who seek to understand particular features of the language. The resulting algorithm VORBS is very effective at slicing picture descriptions, achieving an average 83% reduction in code size. An initial investigation leveraged several off-the-shelf techniques. Shortcomings in this implementation led to the development of a slicing specific matching algorithm. The resulting slices can be used where only a subset of the original image source 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. In this case, 2D PDLs that describe 2D shapes and produce 2D images. The methods could generalize to 3D PDLs and graphical languages that describe 3D shapes and produce 2D images, or that produce 3D images which would require voxel matching. Languages for constructive solid geometry, that describe and produce 3D shapes, could also be addressed. Moving to languages that describe physical motion, such as those that drive 3D printers or robot actuators, would most likely require using simulation to produce digital traces for matching, as recording and matching actual physical motion would be difficult and time consuming. Goal-oriented robot languages, that describe the required state of the world in high level conditions, would require an intelligent ability to perceive achievement of the goals and would be currently beyond our methodology. Finally, an interesting variation is slicing of VLSI circuit diagrams. Because such diagrams have a well defined semantics, VORBS could extract, for example, the register file from a CPU.

## References

- Androutsopoulos, K., Binkley, D., Clark, D., Gold, N., Harman, M., Lano, K., Li, Z., 2011. Model projection: simplifying models in response to restricting the environment. In: Proceedings of the 33rd International Conference on Software Engineering. ACM, pp. 291–300.
- Barpanda, S.S., Mohapatra, D.P., 2011. Dynamic slicing of distributed object-oriented programs. *IET Softw.* 5 (5), 425–433.
- Beck, J., Eichmann, D., 1993. Program and interface slicing for reverse engineering. Proceedings of the 15th International Conference on Software Engineering, pp. 509–518.
- Beszedes, A., Gergely, T., Szabó, Z.M., Csirik, J., Gyimothy, T., 2001. Dynamic slicing method for maintenance of large C programs. In: Proceedings of the 5th Conference on Software Maintenance and Reengineering, pp. 105–113.
- Bhattacharjee, S., Kutter, M., 1998. Compression tolerant image authentication. In: Proceedings of the International Conference on Image Processing, ICIP, October 1998, vol. 1, pp. 435–439.
- Binkley, D., 1998. The application of program slicing to regression testing. *Inf. Softw. Technol.* 40 (11–12), 583–594.
- Binkley, D., Gold, N., Harman, M., Islam, S., Krinke, J., Yoo, S., 2015. ORBS and the limits of static slicing. In: Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015. IEEE Computer Society Press, pp. 1–10.
- Binkley, D., Gold, N., Harman, M., Krinke, J., Yoo, S., 2013. Observation-based Slicing. Research Note RN/13/13. University College London.
- Binkley, D., Gold, N., Harman, M., Krinke, J., Yoo, S., 2014. ORBS: language-independent program slicing. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE November 2014. ACM Press, pp. 109–120.
- Binkley, D.W., 1993. Precise executable interprocedural slices. *ACM Lett. Program. Lang. Syst.* 3 (1–4), 31–45.
- Binkley, D.W., Gallagher, K.B., 1996. Program slicing. In: Zelkowitz, M. (Ed.), In: Advances in Computing, vol. 43. Academic Press.
- Bradski, G., 2000. The OpenCV library. *Dr. Dobbs's J. Softw.* 25 (11), 120–126.
- Choi, J.D., Ferrante, J., 1994. Static slicing in the presence of goto statements. *ACM Trans. Program. Lang. Syst.* 16 (4), 1097–1113.
- Cifuentes, C., Fraboulet, A., 1997. Intraprocedural static slicing of binary executables. In: Proceedings of International Conference on Software Maintenance, pp. 188–195.
- Cimitile, A., De Lucia, A., Munro, M., 1995. Identifying reusable functions using specification driven program slicing: a case study. In: Proceedings of the International Conference on Software Maintenance, ICSM, October 1995, pp. 124–133.
- Cleve, H., Zeller, A., 2000. Finding failure causes through automated testing. In: Proceedings of International Workshop on Automated Debugging.
- De Lucia, A., Fasolino, A.R., Munro, M., 1996. Understanding function behaviours through program slicing. In: Proceedings of the 4th International Workshop on Program Comprehension, pp. 9–18.
- DeMillo, R.A., Pan, H., Spafford, E.H., 1996. Critical slicing for software fault localization. In: Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA), pp. 121–134.
- Ettinger, R., Verbaere, M., 2004. Untangling: a slice extraction refactoring. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD, pp. 93–101.
- Gallagher, K.B., Lyle, J.R., 1991. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.* 17 (8), 751–761.
- Hajnal, A., Forgács, I., 2011. A demand-driven approach to slicing legacy COBOL systems. *J. Softw.: Evol. Process* 24 (1), 67–82.
- Harman, M., 2010. Why source code analysis and manipulation will always be important (keynote). In: Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2010, pp. 7–19.
- Harman, M., Binkley, D.W., Danicic, S., 2003. Amorphous program slicing. *J. Syst. Softw.* 68 (1), 45–64.
- Harman, M., Danicic, S., 1997. Amorphous program slicing. In: Proceedings of the 5th IEEE International Workshop on Program Comprehension, IWPC'97, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.
- Harman, M., Lakhota, A., Binkley, D.W., 2006. A framework for static slicers of unstructured programs. *Inf. Softw. Technol.* 48 (7), 549–565.
- Hierons, R.M., Harman, M., Fox, C., Ouarbya, L., Daoudi, M., 2002. Conditioned slicing supports partition testing. *Softw. Test. Verif. Reliab.* 12, 23–28.
- Horwitz, S., Reps, T., Binkley, D.W., 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.* 12 (1), 26–61.
- Jiang, S., Santelices, R., Grechanik, M., Cai, H., 2014. On the accuracy of forward dynamic slicing and its effects on software maintenance. In: Proceedings of the 114th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28–29.
- Kernighan, B.W., 1981. PIC – a language for typesetting graphics. *SIGPLAN Not.* 16 (6), 92–98.
- Korel, B., Laski, J., 1988. Dynamic program slicing. *Inf. Process. Lett.* 29 (3), 155–163.
- Korel, B., Laski, J., 1990. Dynamic slicing in computer programs. *J. Syst. Softw.* 13 (3), 187–195.
- Krinke, J., 1998. Static slicing of threaded programs. In: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'98, June 1998, pp. 35–42.
- Kusumoto, S., Nishimatsu, A., Nishie, K., Inoue, K., 2002. Experimental evaluation of program slicing for fault localization. *Empir. Softw. Eng.* 7, 49–76.
- Larsen, L.D., Harrold, M.J., 1996. Slicing object-oriented software. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, pp. 495–505.
- Mahajan, S., Halfond, W.G.J., 2015. WebSee: a tool for debugging HTML presentation failures. In: Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation, ICST, April 2015, pp. 1–8.
- McPeak, S., Wilkerson, D. S., Goldsmith, S. Delta: heuristically minimizes interesting files. [delta.tigris.org](http://delta.tigris.org). (accessed 23.03.16.)
- Moigne, J.L., Netanyahu, N.S., Eastman, R.D., 2011. *Image Registration for Remote Sensing*. Cambridge University Press.
- Mund, G.B., Mall, R., 2006. An efficient interprocedural dynamic slicing method. *J. Syst. Softw.* 79 (6), 791–806.
- Paraschenko, O. A., 2011. Delta debugging for  $\text{\LaTeX}$  [uucode.com/blog/2011/04/27/delta-debugging-for-latex/](http://uucode.com/blog/2011/04/27/delta-debugging-for-latex/). (accessed 23.03.16.)
- Parsons-Selke, R., 1989. A graph semantics for program dependence graphs. In: Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages (POPL), Austin, TX, 11–13 January, 1989, pp. 12–24.
- PGF: A Portable Graphics Format for TeX. <http://www.ctan.org/tex-archive/graphics/pgf/>. (accessed 23.03.16.)
- Pluim, J., Maintz, J., Viergever, M., 2000. Interpolation artefacts in mutual information-based image registration. *Comput. Vis. Image Underst.* 77 (2), 211–232.
- Pócsa, K., Biczó, M., Porkoláb, Z., 2005. Cross-language program slicing in the .NET framework. In: Proceedings of the 3rd .NET Technologies Conference, Plzen, Czech Republic, pp. 141–150.

- Podgurski, A., Clarke, L.A., 1990. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.* 16 (9), 965–979.
- Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., Yang, X., 2012. Test-case reduction for C compiler bugs. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012*, pp. 335–346.
- SIPS: Scriptable Image Processing System. <https://developer.apple.com/library/Mac/documentation/Darwin/Reference/ManPages/man1/sips.1.html>. (accessed 23.03.16.)
- Tonella, P., 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Trans. Softw. Eng.* 29 (6), 495–509.
- Ward, M.P., 2003. 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)*, pp. 88–97.
- Weiser, M., 1979. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. University of Michigan, Ann Arbor, MI Ph.D. thesis.
- Weiser, M., 1982. Programmers use slices when debugging. *Commun. ACM* 25 (7), 446–452.
- Weiser, M., Lyle, J., 1985. Experiments on slicing-based debugging aids. In: *Proceedings of the First Workshop Empirical Studies of Programmers*.
- Wheeler, D.A., 2004. SLOC Count User's Guide. [www.dwheeler.com/sloccount/sloccount.html](http://www.dwheeler.com/sloccount/sloccount.html) (accessed 23.03.16.)
- Yoo, S., Binkley, D., Eastman, R.D., 2014. Seeing is slicing: observation based slicing of picture description languages. In: *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014*, pp. 175–184.
- Zeller, A., 1999. Yesterday, my program worked. today, it does not. Why? In: *Proceedings of the 7th European Software Engineering Conference Held Jointly With the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE 1999*, pp. 253–267.
- Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.* 28 (2), 183–200.
- Zhang, X., Gupta, N., Gupta, R., 2007. A study of effectiveness of dynamic slicing in locating real faults. *Empiric. Softw. Eng.* 12 (2), 143–160.
- Zitova, B., Flusser, J., 2003. Image registration methods: a survey. *Image Vis. Comput.* 21 (11), 977–1000.