Towards Learning Visual Semantics

Haipeng Cai
Washington State University
Pullman, Washington, USA
haipeng.cai@wsu.edu

Shiv Raj Pant
Washington State University
Pullman, Washington, USA
shiv.raj.pant@wsu.edu

Wen Li
Washington State University
Pullman, Washington, USA
wen.li@wsu.edu

ABSTRACT

We envision *visual semantics learning* (VSL), a novel methodology that derives high-level functional description of given software from its visual (graphical) outputs. By *visual semantics*, we mean the semantic description about the software's behaviors that are exhibited in its visual outputs. VSL works by composing this description based on visual element labels extracted from these outputs through image/video understanding and natural language generation. The result of VSL can then support tasks that may benefit from the high-level functional description. Just like a developer relies on program understanding to conduct many of such tasks, *automatically* understanding software (i.e., by *machine* rather than by human developers) is necessary to eventually enable fully automated software engineering. Apparently, VSL only works with software that does produce visual outputs that meaningfully demonstrate the software's behaviors. Nevertheless, learning visual semantics would be a useful first step towards *automated software understanding*. We outline the design of our approach to VSL and present early results demonstrating its merits.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; • **Computing methodologies → Scene understanding**;

KEYWORDS

visual semantics, program understanding, computer vision

ACM Reference Format:
Haipeng Cai, Shiv Raj Pant, and Wen Li. 2020. Towards Learning Visual Semantics. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3368089.3417040

1 INTRODUCTION

Understanding the behaviors of software is a fundamental activity in software process. This activity offers the crucial basis for various other tasks such as localizing functionality defects and security vulnerabilities, as well as identifying changes to fix them. This is particularly true during software maintenance and evolution, which is driven by such changes.

Essential for these tasks is the description of the *functional semantics* of a given software application. Just like human developers rely on manually achieved program comprehension for (manually) performing other tasks (e.g., debugging), machines would rely on automatically deduced software understanding in the form of such descriptions for (automatically) performing similar tasks. We see this as the key spirit of *automated software engineering*, where software could be developed and maintained (at least partly) *by machines* instead of human developers. These descriptions would also readily provide an effective means for searching equivalent/similar applications if their semantics descriptions are also available.

Unfortunately, obtaining such descriptions of software semantics is difficult. Manual approaches, such as code reading, is costly, error-prone, and hard to scale, especially with the ever-growing code bases and search sources. Potentially able to overcome these limitations, automated approaches to software understanding would be necessary. However, automating the process of software understanding is challenging. First, software requirements and/or design specifications that describe the software's functionalities would greatly ease the task, yet such documents are usually unavailable. Second, specification mining and recovery techniques exist [31], yet currently their resulting descriptions/representations tend to be too coarse or abstract to be amendable for semantics understanding. Finally, conventional program analysis falls far short of automated semantics inference [21, 22].

We envision *visual semantics learning* (VSL), a novel methodology to automatically understanding the high-level functional semantics of software, hence facilitating the automation of tasks based on the functional semantics. We refer to the description about software behaviors that can be demonstrated through visual outputs as *visual semantics*. Accordingly, software that has visual semantics (i.e., producing visual outputs that meaningfully demonstrate at least part of its behaviors) are referred to as *visual software*. By *visual output*, we mean what a program produces as outputs that consist of visual (graphical) elements, including but *not limited to* GUI elements (e.g., data visualizations such as pie charts and heat maps). Certainly not all software is visual software, and not all of the behaviors of visual software are necessarily manifested through visual outputs. Yet by helping address the task of *automated software understanding* for visual software, VSL would still serve as an essential, first step towards *automated software engineering*.

This paper sketches a VSL approach that leverages advances in image and video understanding (mainly based on deep learning) and natural language processing. Our approach learns visual semantics through recognizing the labels of various visual elements (not only the names of visual objects but also their attributes including spatial relationships with other objects, and scene types) from each visual output as an *image*, as well as inferring temporal relationships and interactions among these elements from the time sequences of multiple visual outputs as *videos*. The approach then generates the semantics description by composing short natural-language sentences from all of the learned visual element labels. This description can be used to support applications that benefit from it (e.g., software classification/search).

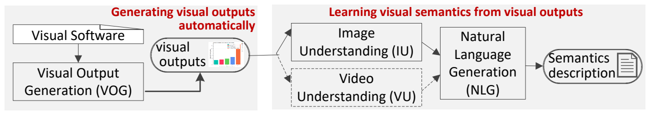
Generating visual outputs automatically
Visual Software
Visual Output Generation (VOG)
visual outputs
Learning visual semantics from visual outputs
Image Understanding (IU)
Video Understanding (VU)
Natural Language Generation (NLG)
Semantics description

Figure 1: An overview of the envisioned approach to VSL.

To demonstrate the benefits of VSL, we implemented an open-source prototype [18] of part of our approach's design using existing techniques for image/video understanding and natural language processing. We then applied it to 16 real-world subject programs from diverse sources that are developed in different programming languages and represent varied application domains. Our early results suggest that our approach did have unprecedented merits in *automatically* learning the visual semantics of the studied subjects (and their executions) with over 90% precision and recall on average for high-level software understanding.

The immediate audience of VSL would be researchers and developers of techniques and tools that support automating software development and maintenance tasks. Through our work, we foresee a new direction in this grand endeavor that exploits latest advances in machine learning and artificial intelligence for automated software understanding.

2 OUR APPROACH TO VSL

We first give a brief overview of the approach we envision and then elaborate each of the key components of the design.

2.1 Overview

As shown in Figure 1, our design includes three major components: *visual output generation* (VOG), *image/video understanding* (IVU), and *natural language generation* (NLG). The main *inputs* include the *visual software* (executable program), optionally with any available *run-time inputs* (e.g., test cases) of the software.

First, the VOG checks if the run-time inputs (when available) are those that exercise all known visual-output-generating APIs (referred to as *visual APIs*), and automatically generates more such inputs if necessary. Then, it runs the program against those inputs to produce visual outputs. Next, the IVU takes the visual outputs and recognizes visual elements, including visual objects (e.g., a curve) and their attributes (e.g., color) as well as (spatial) relationships in each individual visual output as an image, and then produces bags of words (referred to as *visual element descriptors*) that describe each visual element recognized. The IVU further treats the sequence of visual outputs (if exists) as a video to identify temporal relationships across varied visual outputs as additional visual element descriptors. All these descriptors are fed into the NLG module to generate natural-language (NL) sentences (e.g., "the app makes a phone call to a number from a contact list when the user hovers over the number and taps on it") that describe the program's (high-level) functional semantics (referred to as *semantics descriptions*). These resulting descriptions are the *visual semantics* that VSL tries to learn, the *output* of our approach.

2.2 Visual Output Generation (VOG)

How well our approach can learn visual semantics from software (e.g., how much its visual semantics learned covers its full functional semantics) depends on the quality of its visual outputs available for learning. The VOG module aims to exercise as much of the complete visual semantics as possible by first identifying method calls that immediately trigger visual outputs (i.e., visual APIs). Next, the VOG runs the program against the available run-time inputs to check if these inputs exercise all the visual APIs identified. For each exercised visual API, it collects the visual outputs; for any unexercised ones, it automatically generates additional inputs needed through a *targeted input generation* approach.

Specifically, to generate inputs that exercise a visual API *va*, the VOG traverses (interprocedural) control flows from each callsite of *va*, backward to any program entry point, resulting in a set of control flow paths. It then tries to solve the path conditions on each of these control flow paths to either identify the path as infeasible (e.g., the path contains contradicting path conditions), or derive input values from solving the conditions using a constraint solver. Then, the VOG runs the program against these additional inputs to collect more visual outputs.

We note that our approach's eventual scope of outputs is determined by that of the generated visual outputs. The automated software understanding and its application scenarios (e.g., software search) can be either at whole-software level or at class even method levels, as long as the corresponding levels of visual outputs are provided to the rest of the proposed pipeline. We also note that if sufficient visual outputs are supplied as part of its inputs, our approach would be language-agnostic, since otherwise only VOG would be language-dependent—the only module that would require code analysis for input generation.

2.3 Image/Video Understanding (IVU)

The IVU module aims to extract semantic information regarding software functionalities, first from each individual visual output as a single image and then from, if available, the entire (time) sequence of visual outputs as a video, through image and video description generation, respectively. Figure 2 illustrates the extraction process through image understanding based on deep learning (DL). Concrete instances of VSL concepts (e.g., *visual output*, *visual semantics*) are also given in this illustrating example.

To understand each visual output, the IVU focuses on recognizing varied kinds of visual elements, including objects, attributes of each object, spatial relationships among objects, and scene type, using convolutional neural networks (CNNs) which have the advantage (over peer solutions to image understanding) of capturing deep characteristics about visual elements of an image [28]. Given that texts are common in software-produced visual outputs (e.g., button labels like "bar"/"line", menu items, window/dialog title, etc.), the IVU also detects each block of (continuously-spaced) texts as an object using dedicated detectors [39, 40]—the resulting label of such an object is the block of texts itself. Recent advances in computer vision and image understanding [42] for visual relation extraction is utilized to extract relationship information (*visual dependencies* [25]) from the image, which is essential for the NLG to generate meaningful sentences that describe software functionalities.

A key challenge here is to overcome the lack of software-produced visual outputs for training the CNNs. We address this by utilizing the ensemble of existing relevant datasets (e.g., [20, 24, 33]) to extract GUI elements as visual objects. However, these datasets do not include the other necessary types of visual elements (i.e., attributes, relationships, scene type), which we reduce from visual objects using attribute inference techniques [26, 35]. For understanding non-GUI elements, we build the training datasets through a snowballing approach as shown in Figure 2, starting

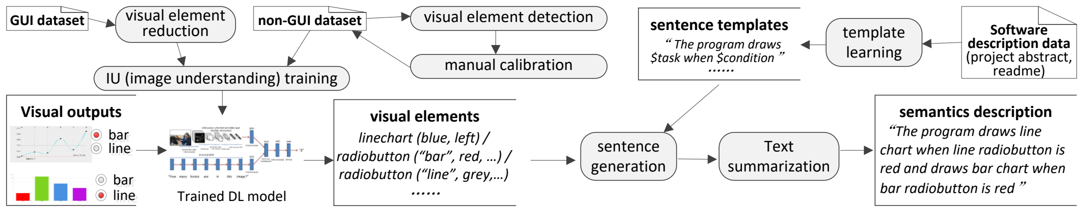
GUI dataset
visual element reduction
non-GUI dataset
visual element detection
manual calibration
IU (image understanding) training
sentence templates
“ The program draws $task when $condition ”
......
template learning
Software description data (project abstract, readme)
Visual outputs
bar
line
bar
line
Trained DL model
visual elements
linechart (blue, left) /
radiobutton (“bar”, red, ...) /
radiobutton (“line”, grey,...)
......
sentence generation
Text summarization
semantics description
“The program draws line chart when line radiobutton is red and draws bar chart when bar radiobutton is red ”

Figure 2: An example illustrating the envisioned visual semantics learning via image understanding.

with example/illustrating programs for graphics and visualization programming (e.g., OpenGL and VTK code examples [32, 38]). First, we curate a small number of samples by running the code examples, and create the visual element labels via manual annotation; next, we use this initial dataset to train an object detector with the capabilities to deduce visual dependencies; then, we use the trained detector to extract the visual elements from additional non-GUI visual outputs, and manually calibrate the results so as to augment the training dataset. Repeating the second and third steps lets us obtain a sizable training dataset for non-GUI visual outputs.

When available, multiple visual outputs (with timestamps) are utilized in two ways. If the outputs form a meaningful sequence (i.e., without abrupt changes) as a video, we exploit the *temporal* structure and attributes of visual objects across the image sequence, in addition to the spatial relationships between visual elements within each image. Specifically, we (1) compute the temporal information from differencing the semantics descriptions between adjacent visual outputs, (2) add the detected changes in and interactions between visual elements to the output of the image-understanding component, and then (3) incorporate the resulting labels (words) to the NLG step. If there are abrupt changes between adjacent visual outputs, video understanding would not apply. In this case, we treat the outputs as individual images and generate the semantics description for each image separately, followed by a text summarization process [41].

2.4 Natural Language Generation (NLG)

The NLG module aims to turn the visual content extracted from the visual outputs into a natural-language-like representation. We pursue this representation mainly because it is a good way to describe the structural and semantic relationships among the visual elements of the visual outputs, whereas these relationships are an essential part of functional (visual) semantics of software.

Specifically, with the list of unstructured words resulting from the IVU step, we generate the target representation using a hybrid approach combining template-based and statistical approaches, as illustrated in Figure 2. We adopt part of the template-based idea by filling sentence templates with the labels (as nouns) indicating visual elements and visual dependencies among the elements (as conjunctions, prepositions, etc.). To mitigate the tedious and unscalable nature of the template-based approach, we then exploit the spirit of statistical approaches at the same time: automatically learn sentence templates from a limited training dataset, using a state-of-the-art statistical framework (e.g., [29]). To that end, we use project descriptions and README files from top-quality open-source project repositories (e.g., on GitHub) to build the initial training dataset for the template learning purpose.

A main challenge with this approach to NLG seems to lie in its limited accuracy in practice, due to the potential failure of underlying IVU techniques to accurately detect the visual elements

Table 1: Subject programs used in our evaluation

Subject	Language	Ground-truth functionality
MPAndroidChart [1]	Java	Produces various kinds of charts
GeometricObjects [2]	Python	Produces various geometric objects
Python-QRcode [14]	Python	Generates QR code
QR code [15]	PHP	Generates QR code
StackedBarGraph [12]	Python	Produces a barchart
BarchartDemo [13]	Python	Produces a barchart
Circle [3]	C++	Generates a circle
Matplotlib-circle [4]	Python	Generates a circle
Triangle [5]	C++	Produces a triangle
Streamplot [6]	Python	Produces a streamplot
SIP Caller [7]	Java	Makes call to SIP numbers
Emerald Dialer [16]	Java	Makes phone call
AppAppCall [8]	Java	Makes app-to-app call
Piechart [9]	Python	Procudes a piechart
Piechart2 [10]	Python	Produces a piechart
Rectangle [11]	C++	Generates a rectangle

and/or that of the underlying natural-language generation techniques to accurately produce fluent and grammatically correct sentences. However, as mentioned earlier, VSL does not target human developers and help them understand visual semantics (which they would rather directly obtain by viewing the visual outputs generated). Instead, the users of the visual semantics descriptions are automated techniques serving application tasks based on software understanding (e.g., automated code reuse, fault diagnosis, software search, etc.). For these automate applications, the representation does not have to be perfectly correct in terms of natural language grammar or being fluently human-readable—the more important requirement is that it will maintain the visual elements and their key relationships.

3 EVALUATION

For a preliminary validation of our approach, we partly implemented the envisioned design using simplified choices for its various technical components. For the IVU module, we trained a CNN model [30] using different types of images of non-GUI objects (e.g., simple geometric shapes such as circles, triangles, charts etc.) and cropped images of standard GUI elements (buttons, checkboxes, etc.) collected from online resources. With such a trained IVU module, the prototype only recognizes object labels from individual images separately at this stage. For the NLG component, we simply adopted a template-based sentence generation method, using templates manually created based on IVU-produced object descriptors (e.g., “this program <verb> <object>"). The individual sentences generated were then summarized to construct concise semantics descriptions using NLTK [17].

We applied this early prototype to 16 real-world applications, as summarized in Table 1. They were obtained from diverse sources (github, vtk.org [37], F-droid [27], and matplotlib [32]) and represented different programming languages. We manually exercised each application to generate a single (for 11 subjects) or multiple (for the other 5) visual outputs, including GUI and non-GUI ones. The subjects also included those of similar functionalities to facilitate our evaluation. The ground-truth functionalities were manually derived and used to compute the precision and recall of the prototype in generating semantics descriptions.

Effectiveness. We considered the semantics description generated by the prototype for an application a true positive if the description did capture the application's functionalities demonstrated in the given visual outputs, and a false positive otherwise. The prototype achieved 96% precision and perfect recall for the 16 such descriptions it produced—the imprecision was due to failures in recognizing the visual objects for two subjects (MPAndroidChart and GeometricOjbects) as a result of insufficient CNN training.

Efficiency. The main cost of our approach generally lies in that for training and loading the IVU and NLG models, and depends on the size of these datasets as well as the time for generating the visual outputs. For our study in particular, generating all models needed took no more than 3 minutes. Beyond this, the time for the prototype to generate the semantics description for each subject was negligible. Since the visual outputs were manually produced for now, the VOG cost was not counted here.

4 RELATED WORK

The prior work that is most related to ours is ReDraw, a GUI code generation technique for Android apps as presented in [33]. Similar to our work, this technique recognizes GUI elements of programs through computer vision based on deep learning models (CNNs). Meanwhile, the VSL concept and our approach to VSL are new with respect to this work in multiple ways. First, ReDraw aims to generate GUI implementation (code), while we envision VSL to understand the high-level program functionalities from visual outputs by generating natural-language descriptions of the functionalities. These outputs from which VSL learns include but are not limited to GUI elements. Second, while ReDraw helps Android app developers sketch up GUI code, VSL targets automated software tools as users that leverage automatically obtained understanding of software in general (not mobile apps only). Third, ReDraw's visual object recognition was limited to individual GUIs as static images, while our approach also learns visual semantics from time sequences of visual outputs as videos.

Like other prior work on GUI prototyping that generates GUI design [23] and code [20, 34] skeleton from a given GUI image or screenshot, our approach leverages data-driven techniques for visual element recognition. The semantics descriptions resulting from VSL can support software search (e.g., through natural language matching). Yet this VSL application would focus on finding existing software that match functional semantics expressed by visual outputs according to user queries, rather than matching a particular kind of visual outputs (GUIs) themselves—again, our visual outputs are generally defined, not limited to GUIs.

Current work on GUI search aim at a more immediate step from a GUI skeleton to the code that implements the GUI by searching an existing code repository (rather than generating the code as GUI prototyping does) [19, 36]. In comparison to a VSL-based application search technique, the prior work targeted searching code that implements given GUIs themselves (but not what they mean/do—their semantics).

Compared to the rich body of work on program understanding by developers, we target automated software understanding by machines. And we are not aware of prior work on understanding functional semantics from software visual outputs.

5 CONCLUSION & FUTURE WORK

We presented the design of a novel approach to automating software understanding via visual semantics learning. Our preliminary empirical results demonstrated promising prospects of the proposed approach. An immediate next step is to realize our full design hence evaluate the approach systematically. We envision this work to stimulate a new direction in leveraging artificial intelligence for automated software engineering.

REFERENCES

[1] 2018. https://github.com/PhilJay/MPAndroidChart.
[2] 2018. http://shorturl.at/bdijs.
[3] 2018. https://vtk.org/Wiki/VTK/Examples/Cxx/GeometricObjects/Circle.
[4] 2018. http://shorturl.at/asyZ6.
[5] 2018. https://vtk.org/Wiki/VTK/Examples/Cxx/GeometricObjects/Triangle.
[6] 2018. http://shorturl.at/fhsT9.
[7] 2018. https://f-droid.org/en/packages/org.whitequark.sipcaller/.
[8] 2018. https://github.com/sinch/app-app-calling-android.
[9] 2018. http://shorturl.at/JKMUZ.
[10] 2018. http://shorturl.at/fIL12.
[11] 2018. https://github.com/softvar/OpenGL/blob/master/primitives/rectangle.cpp.
[12] 2018. Stacked Bar Graph. http://shorturl.at/jLQX4.
[13] 2018. Stacked Bar Graph. http://shorturl.at/cpR58.
[14] 2019. https://github.com/lincolnloop/python-qrcode.
[15] 2019. https://github.com/endroid/qr-code.
[16] 2019. https://f-droid.org/en/packages/ru.henridellal.dialer/.
[17] 2019. https://www.nltk.org/.
[18] 2020. Prototype. https://github.com/Daybreak2019/vissem.
[19] Farnaz Behrang et al. 2018. GUIfetch: supporting app design and development through GUI search. In *MOBILESoft*. 236–246.
[20] Tony Beltramelli. 2018. pix2code: Generating code from a graphical user interface screenshot. In *EICS*. 1–6.
[21] Haipeng Cai et al. 2014. SensA: Sensitivity Analysis for Quantitative Change-impact Prediction. In *SCAM*. 165–174.
[22] Haipeng Cai et al. 2016. Prioritizing Change Impacts via Semantic Dependence Quantification. *IEEE Transactions on Reliability (TR)* 65, 3 (2016), 1114–1132.
[23] Chunyang Chen et al. 2018. From UI design image to GUI skeleton: a neural machine translator to bootstrap mobile GUI implementation. In *ICSE*. 665–676.
[24] Biplab Deka et al. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *UIST*.
[25] Desmond Elliott et al. 2013. Image description using visual dependency representations. In *EMNLP*. 1292–1302.
[26] Desmond Elliott et al. 2015. Describing images using inferred visual dependency representations. In *ACL-IJCNLP*. 42–52.
[27] f droid.org. 2018. F-droid. https://f-droid.org/en/.
[28] Yanming Guo et al. 2016. Deep learning for visual understanding: A review. *Neurocomputing* 187 (2016), 27–48.
[29] Ravi Kondadadi, Blake Howald, and Frank Schilder. 2013. A statistical NLG framework for aggregated planning and realization. In *ACL*. 1406–1415.
[30] Yann LeCun et al. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86, 11 (1998), 2278–2324.
[31] David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. 2011. *Mining software specifications: methodologies and applications.* CRC Press.
[32] matplotlib.org. 2018. Sample Python programs. https://matplotlib.org/.
[33] Kevin Moran et al. 2018. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *TSE* (2018).
[34] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui. In *ASE*. 248–259.
[35] Luis Gilberto Mateos Ortiz et al. 2015. Learning to interpret and describe abstract scenes. In *NAACL-HLT*. 1505–1515.
[36] Steven P Reiss, Yun Miao, and Qi Xin. 2018. Seeking the user interface. *Automated Software Engineering* 25, 1 (2018), 157–193.
[37] vtk.org. 2018. Sample VTK programs. https://vtk.org/.
[38] vtk.org. 2020. sample visualization programs. https://gitlab.kitware.com/vtk/vtk/tree/master/Examples.
[39] Qixiang Ye et al. 2005. Fast and robust text detection in images and video frames. *Image and vision computing* 23, 6 (2005), 565–576.
[40] Xu-Cheng Yin et al. 2013. Robust text detection in natural scene images. *TPAMI* 36, 5 (2013), 970–983.
[41] Yongjian You et al. 2019. Improving Abstractive Document Summarization with Salient Information Modeling. In *ACL*. 2132–2141.
[42] Hanwang Zhang et al. 2017. Visual translation embedding network for visual relation detection. In *CVPR*. 5532–5540.