Spotting problematic code lines using nonintrusive programmers' biofeedback

Ricardo Couceiro
CISUC, University of Coimbra
Coimbra, Portugal
rcouceir@dei.uc.pt

Raul Barbosa
CISUC, University of Coimbra
Coimbra, Portugal
rbarbosa@dei.uc.pt

João Durães
CISUC, Polytechnic Institute of Coimbra
Coimbra, Portugal
jduraes@isec.pt

Gonçalo Duarte
CISUC, University of Coimbra
Coimbra, Portugal
duarte.1995@live.com.pt

João Castelhano
ICNAS, University of Coimbra
Coimbra, Portugal
joaocastelhano@uc.pt

Catarina Duarte
ICNAS, University of Coimbra
Coimbra, Portugal
catarinaduarte86@gmail.com

Cesar Teixeira
CISUC, University of Coimbra
Coimbra, Portugal
cteixei@dei.uc.pt

Nuno Laranjeiro
CISUC, University of Coimbra
Coimbra, Portugal
cnl@dei.uc.pt

Júlio Medeiros
CISUC, University of Coimbra
Coimbra, Portugal
juliocordeiromedeiros@gmail.com

Paulo de Carvalho
CISUC, University of Coimbra
Coimbra, Portugal
carvalho@dei.uc.pt

Miguel Castelo Branco
ICNAS/CIBIT, University of Coimbra
Coimbra, Portugal
mcbranco@fmed.uc.pt

Henrique Madeira
CISUC, University of Coimbra
Coimbra, Portugal
henrique@dei.uc.pt

***Abstract*— Recent studies have shown that programmers' cognitive load during typical code development activities can be assessed using wearable and low intrusive devices that capture peripheral physiological responses driven by the autonomic nervous system. In particular, measures such as heart rate variability (HRV) and pupillography can be acquired by nonintrusive devices and provide accurate indication of programmers' cognitive load and attention level in code related tasks, which are known elements of human error that potentially lead to software faults. This paper presents an experimental study designed to evaluate the possibility of using HRV and pupillography together with eye tracking to identify and annotate specific code lines (or even finer grain lexical tokens) of the program under development (or under inspection) with information on the cognitive load of the programmer while dealing with such lines of code. The experimental data is discussed in the paper to assess different alternatives for using code annotations representing programmers' cognitive load while producing or reading code. In particular, we propose the use of biofeedback code highlighting techniques to provide online programmer's warnings for potentially problematic code lines that may need a second look at (to remove possible bugs), and biofeedback-driven software testing to optimize testing effort, focusing the tests on code areas with higher bug probability.**

Keywords: *software faults, eye tracking, pupillography, heart rate variability, mental effort, cognitive overload, human error, code annotation.*

I. INTRODUCTION

Knowing that software development is a human intensive and error prone intellectual task, it is surprising the relatively low number of research works that study the problem of software faults from an individual (i.e., programmer, tester, etc.) perspective, taking software faults (i.e., bugs) as the consequences of individual human errors in different steps of the software development process. Software engineering in general, and empirical software engineering more specifically [1], have studied human factors in the software development process, including quality aspects related to software faults. But most of the works in such disciplines focus on human factors related to behavior, attitudes, and even cultural aspects in software development communities, as well as organizational issues related to group dynamics [2]. The root causes of human error mechanisms, particularly in an individual programmers/testers' perspective, have been largely ignored, and none of the software development methodologies and tools available today take into account (in a direct form) potential human error causes that may affect programmers, testers, and software engineers during the development of software artifacts.

This paper proposes the use of nonintrusive programmers' biofeedback to annotate the code produced or inspected by individual programmers, in order to tag code lines (and lexical tokens in each line) with information related to the programmers' cognitive load (and possibly other emotional states such as mental effort, stress level, attention level, mental fatigue,...). The assumption is that these code annotations with data about programmers' cognitive load can be used to identify potential conditions that may lead programmers to make bugs, or may cause bugs to escape human attention. Evidence from cognitive human error theories and models [3] and, particularly, recent cognitive taxonomy on human error causes for software defects resulting from field studies [4, 5] support the assumption that, for instance, mental overload and attention slips are important causes for software bugs. This is precisely the information we propose to associate to source code lines and tokens through biofeedback code annotation.

It is a well-known fact that peripheral physiological responses driven by the Autonomic Nervous System (ANS) can be correlated to human cognitive load, mental effort, and attention level while performing specific tasks. Furthermore, these psychophysiological manifestations can be captured by nonintrusive sensors compatible with programmers' code development activities. In fact, there are a variety of commercial wearable devices that can monitor ANS driven response such as heart rate variability (HRV), breathing rhythm, electrodermal activity (EDA), and eye tracking with pupillography. Very recent studies, published in 2019, have shown that HRV [6] and pupillography [7] can be used to assess programmers' mental effort, allowing the automatic identification of code units considered by code developers as more difficult and potentially more susceptible to contain bugs.

Building on these recent results [6, 7], the present paper proposes the use of eye tracking, together with HRV and

pupillography, as the basic set of nonintrusive sources of programmers' biofeedback to allow the accurate identification of code lines (and even lexical tokens inside code lines) that correspond to mental effort peaks, **laying the ground for fine grain biofeedback code annotation**. The nonintrusive features of the proposed sensors set are a key aspect to allow practical application of the proposed biofeedback code annotation in real world software development environments. In fact, HRV signals can be gathered using modern smart watches, and pupillography and eye tracking can be done using desktop eye tracking devices that do not even require any physical contact with the programmer.

This proposed approach is presented and discussed in the paper through the analysis of data obtained from an extensive experiment involving 30 programmers equipped with a complete set of sensors to record ANS response while doing code comprehension tasks similar to the ones performed in traditional code inspections. The central goal of such experiment is to show the feasibility of code line and token discrimination, as basic resolution grain for biofeedback code annotation. Based on the experimental results, the paper anticipates the key aspects of the proposed framework to use biofeedback code annotation in real software development environments and proposes the concepts of **biofeedback code highlighting** to allow online warning of programmers, calling their attention to code lines with higher probability of containing bugs, and **biofeedback-driven software testing**, to concentrate the testing effort in potentially problematic code areas.

The structure of the paper is as follows: next section presents the background of our research and related work, Section III describes the experiment design and protocol, Section IV presents the methodology used in the data analysis, Section V discusses the results, Section VI proposes the code biofeedback annotation framework and examines possible utilization of such proposal, Section VII presents the threats to validity, and Section VIII concludes the paper.

II. BACKGROUND AND RELATED WORK

Despite many years of accumulated experience of software programming, and the large body of knowledge concerning training and development methodologies, software faults continue on the rise, especially because software has grown in complexity and size (e.g., a modern high end car has more than 100 millions lines of code). Field studies on real bugs found in deployed software, covering different projects, types of systems, and built according to different methodologies, have shown that the types of bugs found are similar [8, 9, 10], and that most of the bugs follow a relatively small number of bug types. These findings show that software developers (especially the programmers) tend to err in similar ways, no matter the software methodology, programming language, and development environment, suggesting that a finite (and not very large) number of human error mechanisms are at the very origin of most software bugs.

The academia has recently realized the potential of exploring cognitive psychology, neuroscience, and health informatics fields to help understanding the complex nature of software programming as human intellectual task. The general goal behind this interdisciplinary research trend is, obviously, to improve the software development in large, which touches many aspects of the development process. In particular, software reliability and the cognitive processes related to code comprehension, which is implicitly related to the problem of software faults, have been addressed in recent interdisciplinary works. The next paragraphs briefly summarize these works.

Classic theories and models on human errors [3] have been adapted to software development process [4, 5, 11] to help defining error taxonomies and propose strategies to reduce the number of bugs in the software. The idea is to understand the way humans make mistakes when developing software in order to devise general strategies to prevent or mitigate bugs. This research line, however, does not take into account the individual aspects (i.e., related to specific programmers and their context) that may cause software faults.

Another recent research line consists of researching the brain and neural mechanisms related to programming tasks such as code comprehension and code inspection. This approach merges computer science with neurosciences and relies on heavy instrumentation such as EEG (electroencephalography), fNIRS (near field infrared spectroscopy), and fMRI (functional magnetic resonance imaging) to collect data on brain activity in reading and code inspecting tasks. The work presented in [12] studied the mental effort involved in analyzing and comprehending source code. The work in [13] characterizes the brain mechanisms involved in understanding natural language texts and in source code, comparing the brain mechanisms involved in each case. Siegmund et al. [14, 15] identify specific regions of the brain involved in code comprehension and syntax error identification, specifically the regions of language processing, working memory and attention.

The specific study on the programmer's brain when dealing with the task of finding bugs in software was addressed in [16]. That study suggests that the activity levels of the insula region of the brain are directly related to the quality of the bug detection, establishing a direct relationship of a brain signal with a software development skill, and opening the possibility of using the activity of that brain structure as predictor of accuracy of bug finding tasks.

Heavy instrumentation such as fMRI cannot be used in a real industry software development setting. However, these works open an import research avenue on the relationship between the brain activity in specific software development tasks and error prone scenarios, and the physiological manifestations driven by the ANS that can be captured by nonintrusive and wearable devices such as the ones providing HRV and pupillography data. This is essential to improve data processing models to reduce noise and mitigate spurious signals not directly related to the software development tasks.

The use of HRV as programmers' mental effort indicator have been demonstrated in [6], showing that HRV can be used to differentiate software code units considered by the programmers difficult and complex (i.e., require high levels of mental effort) from other code units that the programmers consider simple. Müller et al. [17] proposed the used of HRV as a predictor of the quality of code made by programmers, suggesting that HRV can be used to guide the testing effort.

The eye pupil has long been recognized as a good indicator of mental and cognitive effort as shown by the early works of [18, 19], just to name a few examples. The opening and closing of the pupil is controlled by signals from both the sympathetic and the parasympathetic elements of the autonomic nervous system [20, 21], making its size variations

a reliable indicator of the mental state and stress level of the person. Several recent studies explore pupillography and show that pupil size variations are related to mental and cognitive load [19, 20, 22].

Researchers also found that the spectral analysis of the signal related to pupil diameter (PD) variation provides useful information regarding cognitive load and fatigue states. The work described in [23] used PD variation with frequency below 0.8 Hz to characterize sleepiness states, and the work in [24] found a relationship between the increase of spectral density of the PD variation signal within 0.1–0.5 Hz and 1.6–3.5 Hz and the increase of the difficulty of calculation tasks. The ratio of the low frequency (0.05-1.6 Hz) and the high frequency (1.6-4Hz) bands of the PD variation signal spectrum also provides information regarding cognitive load. In [25] it is shown that low frequency domain (below 0.8 Hz) of pupil variations allow measuring fatigue and sleepiness, while [26] suggests the ratio 0 Hz–1.6 Hz to 1.6 Hz – 4 Hz is not sensitive to changes in luminance, which means that variations in this ration are more related to the mental load than to the surrounding environment.

More specific to the software development context, [7] has shown that pupillography can be used as an indicator of code complexity and mental effort as perceived by programmers, which clear connection to error prone scenarios.

Eye tracking devices have been used to research diverse aspects of software engineering [27]. A set of eye tracking metrics applied to software engineering was proposed in [28]. Moreover, eye tracking was proposed for several tasks of software traceability [29], programming education and training [30], program comprehension analysis in programmers with dyslexia [31], among many others.

Eye tracking is proposed in this paper in a tightly coupled association with HRV and pupillography, aiming at a preliminary exploitation of two aspects of such association: **a)** the use of eye tracking to localize **in space** (i.e., code lines and tokens) and **in time** the indications of programmers' mental effort provided by HRV and pupillography and **b)** consolidate HRV and pupillography indications with eye tracking specific metrics (e.g., areas of interest with longer fixation times, revisits, speed of code analysis, etc.) that provide additional information on programmers' mental effort.

III. EXPERIMENT DESIGN AND PROTOCOL

The goal of our experiment is to investigate the possibility of using eye tracking, in strict association with HRV and pupillography, to allow fine grain code annotation at code line and token level. These annotations provide information about the programmers' mental effort and cognitive states while handling (writing, reading, changing) such code lines and tokens, and will be used to identify specific code points that may have higher probability of having bugs.

The observational study designed consists of having a group of 30 programmers that perform reading and code understanding tasks in 3 programs written in Java, having different sizes and complexity. The ANS activity of the programmers during the code comprehension tasks, particularly the sympathetic and parasympathetic activity imbalances induced by the diverse levels of mental effort required to understand the different areas of the programs, is gathered using a set of sensors that includes HRV (using a BiosignalsPlux toolkit) and an eye tracker equipped with pupillography (using a SMI Senso Motoric Instruments eye tracker). The signals from the sensors are collected using a common time base to assure synchronization and to allow accurate cross analysis.

The group of 30 Java programmers includes 24 male and 6 female, with ages of 24.4 ± 6.18 years. These programmers participated in a volunteer basis and have been selected from a larger pool of candidates through an interview-based screening process focused on the assessment of their Java programming skills. The goal of this screening was to avoid selecting Java beginners, since the experiment is intended to reproduce a professional programming context as much as possible. The 30 participants were classified in three skill levels as follows: Intermediate: 12 participants; Advanced: 14 participants, and Expert: 4 participants.

An important aspect carefully handled in the study preparation was to inform the participants they are not under evaluation in any way, to reduce the effect of being "watched". In any case, the way pupillography and HRV features are extracted are essentially differential, using baseline activities such as reading a text or doing nothing during 30 seconds as a reference, so the effect of being "watched" is not significant.

The study was approved by the Ethical Commission of the Faculty of Medicine of the University of the University of Coimbra, in accordance with the Declaration of Helsinki, and all the typical procedures for studies involving human subjects were observed, namely all participants provided written informed consent and all the date was anonymized. The anonymized raw data collected in this study are available to the research community upon request to the authors.

The programs used in the tasks (similar to typical code inspections) are 3 small programs written in Java, herein named as C1, C2 and C3. The programs have been carefully designed to keep consistency in the programming style and to avoid extra difficulties, such as heavy math, not directly related to the code complexity. The following bullets briefly characterize each program:

- **C1** is a very simple program that counts the number of values existing in an array that fall within a given interval using a straightforward loop. The program has 13 code lines and the cyclomatic complexity Vg =3;
- **C2** performs the multiplication of two numbers using the basic algorithm for arithmetic multiplication. First the program converts the string input parameters into byte arrays. Next, a straightforward multiplication is implemented where every digit from one number is multiplied by every digit from the other number, from right to left. The program has two functions with a total of 42 code lines and Vg = 3 and Vg = 6, respectively.
- **C3** is a search program that finds the largest occurrence of an integer cubic array inside a larger cubic array. This programs has many nested loops and, therefore, has a high cyclomatic complexity. It has 49 lines of code and Vg = 14.

C1 represents a very simple and linear code. C2 and C3, although having a similar number of code lines, C2 was expected to be simpler and easy to understand than C3, since Vg of C2 is much lower than Vg of C3. In short, our intention was to have a very simple program (C1), an intermediate complexity one (C2) and a relatively complex program (C3).

The protocol defined for the experiment define strictly similar conditions to all the participants using a controlled environment without distractions, noise or the presence of people unrelated to the experiments. The steps of the protocol, performed on the screen of a laptop, are the following:

1. **Baseline activity**: participants look at an empty grey screen with a black cross in its center for 30 seconds. Baseline cross is also useful for eye tracking fine calibration (after the standard eye tracking calibration).
2. **Reference activity**: participants read a text in natural language for no more than 60 seconds.
3. **Baseline activity**: participants look at an empty grey screen with a black cross for 30 seconds.
4. **Code comprehension**: participants read and try to fully understand the Java programs (C1, C2, C3). This step last 10 minutes maximum for each program.
5. **Baseline activity**: participants look at an empty grey screen with a black cross for 30 seconds.
6. **Survey 1**: NASA-TLX[1] to assess the subjective mental effort perceived by each participant in the code comprehension.
7. **Survey 2**: check if each participant understood each program through a set of simples questions about the program goal and structure.

This protocol was repeated 3 times for each participant, using a different program (C1, C2, C3) in step 4 for each time.

A total of 90 data sets have been collected in this study, corresponding to the 30 volunteers times 3 tasks (C1, C2, C3) per volunteers. However, 21 data sets have been excluded later on (4 C1 pieces, 8 C2 pieces and 9 C3 pieces) due to a systematic eye tracking calibration problem that was only detected after the data collection experiments. Thus, the number of useful data sets is 69.

IV. METHODS AND ANALYSIS PROCEDURE

The data obtained in the experiment was explored in two preferential ways (nevertheless, the data set is so rich that it is possible to perform several other types of analysis):

1) The changes in the heart rate variability (HRV) and in the pupil diameter (PD) during code comprehension tasks were analyzed to identify moments that correspond to peaks in the mental effort of the programmer, and these points were mapped to the corresponding code locations using the eye tracker information. This is the natural way we expect to use the code annotation, since the actual information that should be conveyed in the annotations is related to the mental effort and cognitive state of the programmer while dealing with specific program lines/tokens.

2) Analysis of specific code regions identified as being potentially critical regions. We used two methods to define possible critical regions to study in more detail: **a)** regions (i.e., specific code lines) pointed by a small group of Java experts as potentially difficult for program comprehension, and **b)** regions of interest identified using eye tracking metrics, mainly obtained through clustering techniques over space and time domains. To perform this second analysis we look at the coordinates in the screen to which the volunteer is looking at, which is called point of regard (POR), in order to identify clusters of data corresponding to regions potentially critical (e.g., because the volunteer spent long time analysing the code in such regions, recurrentely revisit such regions, etc) and the changes in PD and HRV were analyzed within these regions.

The next subsections presents the methods used to perform the analysis. Subsection A describes the methods used to locate critical regions using eye tracking and subsections B and C present the methods used to preprocess HRV and PD signals and to analyze these two physiological signals.

A. Identification of critical code regions

In order to identify the regions that are potentially interesting, the point of regard (POR) coordinates, i.e. the coordinates in the screen to which the volunteer is looking at, for each task were analyzed within the time span of the code analysis. The main idea here is to 1) identify the clusters of data corresponding to areas of interest and 2) identify, within these clusters the ones that correspond to time frames of cognitive load increase.

1) Pre-processing of POR data

To achieve these objective the first step was the pre-processing of the POR data. **Firstly**, the x-y coordinates of the POR were down sampled to 10 Hz. **Secondly**, although there is a calibration step of the eye tracker prior to the beginning of each task, it was observed that some POR data was still displaced. In order to correct this displacement, the POR data was shifted with an increment corresponding to the difference (in the x and y axis) between the median of the POR values during the fixation of the calibration cross and the actual calibration cross. The **third** step of the pre-processing stage was the removal of the POR values that laid outside the code snipped area. This was accomplished by defining the contour of the code snippet (with an offset) and identifying all POR values inside this region.

2) Clustering

The identification of the clusters was performed using the Density-based spatial clustering algorithm using a three dimension feature space.

The features used for clustering are: 1) time instants; 2) y-coordinates of the POR and; 3) distance between consecutive POR points. The rationale behind the third feature is that larger distances between consecutive POR points correspond to faster velocities (since the time span is constant), which intuitively suggests faster analysis (reading) of the corresponding code region).

The parameters used in the clustering algorithm were:

- Neighborhood search radius: twice the length of a code line in pixels plus the gap between the code lines, i.e. 45 pixels
- Minimum number of neighbors: 2 time the sampling frequency, i.e. 20 neighbors

3) Identification of potentially critical regions

[1] NASA human factors web page: https://humansystems.arc.nasa.gov/groups/TLX/

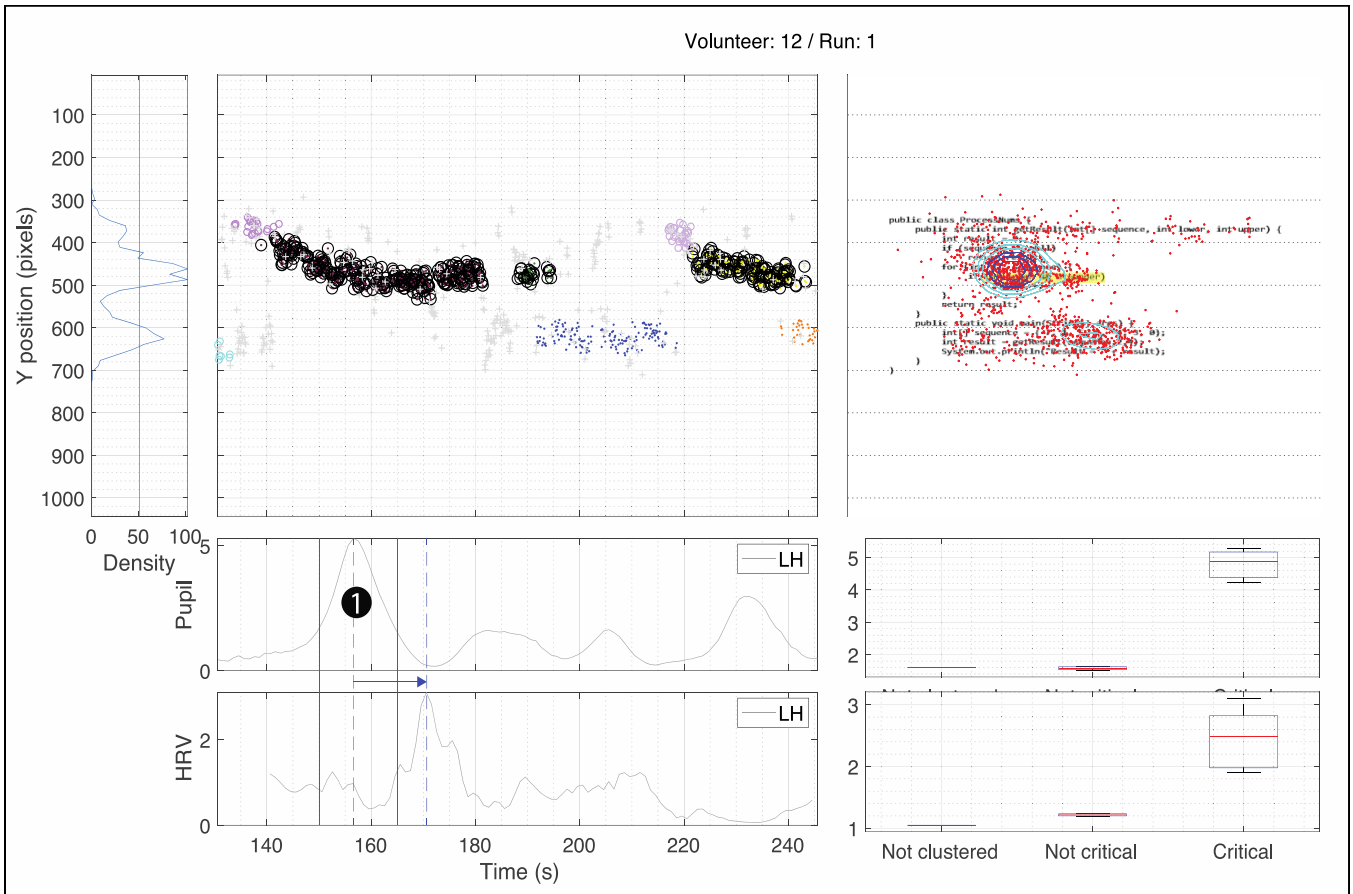

Figure 1 - Eye tracker and HRV results achieved in experience for Subject 12, code C1

Upper row/left – density of gaze line number; Upper row/middle – clusters of gaze line over time and gaze velocity; Upper row/right – red dots: geometrical distribution of gaze points superimposed on code during experience; yellow rectangle potentially critical regions of inspected code manually annotated by experts.
Middle row/left – Extracted pupilography feature LH; Middle row/right – box plot of pupilography outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions.
Lower row/left – Extracted HRV feature LH; Lower row/right – box plot of HRV outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions

The identification of the potentially critical regions was done through the calculation of a Bivariate histogram of the POR x-y coordinates. The bins edges were defined so that each row of bins has the size of a code line plus the gap between the code lines (i.e. 25 pixels). The density of POR points in each row was calculated and rows with a density greater than 50% of the maximum row density were identify as critical code areas. The clusters that fall inside the identified code areas were taged as selected clusters, while the remaining clusters were identified and not selected clusters. All the remaining POR points were identified as cluster outliers.

B. *Preprocessing of HRV and pupillography signals*

For HRV we used as preprocessing a standard Pan-Thompkins segmentation algorithm to extract the RR intervals. Concerning pupillography, we preprocessed the PD signal from the left eye. All intervals labeled by the eye tracker as having invalid pupil diameter (PD) values (Δ_{inv}), the adjacent intervals, i.e. $PD\left([\Delta_{inv} - 0.1, \Delta_{inv} + 0.1]\right)$, and other spurious values were considered as inaccurate. These values identified as inaccurate were interpolated using a shape-preserving piecewise cubic interpolation. The resulting PD signal was down sampled to 20 Hz, reducing the data size considerably while preserving the frequency contents for the further analysis (0 to 10 Hz).

In order to reduce the influence of the artifacts related to eye blinks and other external factors, an algorithm for filling in missing data based on iterative Singular Spectrum Analysis [32, 33, 34] was used in this study to reduce their impact in the extraction of the frequency domain features [35].

Finally, the PD time series was high-pass filtered with a very low cutoff frequency (4×10^{-4} Hz) in order to minimize the effects of medium-term nonstationary within the time interval under analysis [36].

C. *HRV and pupillography analysis*

We concentrate our analysis on the HRV and pupillography features that previous studies [6, 7] have identified as the best to indicate mental effort in code comprehension tasks.

For both pupillography and HRV, we used the LH feature that corresponds to the ratio between the low frequency (0.04-0.15 Hz) and high frequency (0.15-0.4 Hz) spectral power of the pupil diameter variability/ ECG's RR interval variability assessed using a 30 seconds and a sliding window shifted with 1 sec increments. In both features, power spectrum was computed using Burg's autoregressive power spectrum approach [37]. The LH ration in both physiological signals capture the imbalance between the sympathetic and parasympathetic of the ANS which is known to be highly sensitive to cognitive stress [26].

V. RESULTS AND DISCUSSION

Previous studies [6, 7] have shown that it is possible to differentiate (using HRV and pupillography) the average level of cognitive load of programmers when performing code comprehension tasks in program units with different complexity levels. Although providing useful indications, the results presented in [6, 7] are based on the average cognitive load, which cannot be used to annotate code in a detailed level. Obviously, only relatively detailed annotations, such as annotations at code line level or token level could provide interesting programmers' biofeedback information. The analysis performed in this section is meant to investigate the possibility of performing detailed code annotation through the association of eye tracking to the HRV and pupilography.

A first aspect that should be discussed is the perceived complexity of the 3 programs used in the experiment. As can be observed in Table 1, code C2 and code C3 were perceived by participants as having similar complexity using the NASA-TLX tool. In this table, objective evaluation OE captures the average correctness of the answers related to the understanding of the programs (0 means complete failure in understanding the code and 1 a complete success). Mental effort (ME), Time Pressure (TP) and Discomfort (DI) were collected with a NASA-TLX enquiry (second survey). All values are from 0 to 1.

In fact, although codes C2 and C3 have quite different cyclomatic complexity (Vg), the subjective perception of participants obtained through the NASA-TLX survey shows that programmers consider that both codes have similar complexity. Although it is not entirely new that complexity metrics do not correlate with human perceived code complexity in many situations, this result is an impressive example that Vg is far from being an accurate indicator of code complexity in a programmers' perspective.

Figure 1 to Figure 4 depict four distinct results achieved during the outlined code comprehension experience for distinct users with different proficiency level; the results shown in Figure 2 correspond to an expert level participant whereas Figure 1, Figure 3, and Figure 4 were obtained using participants that experienced more difficulty in understanding

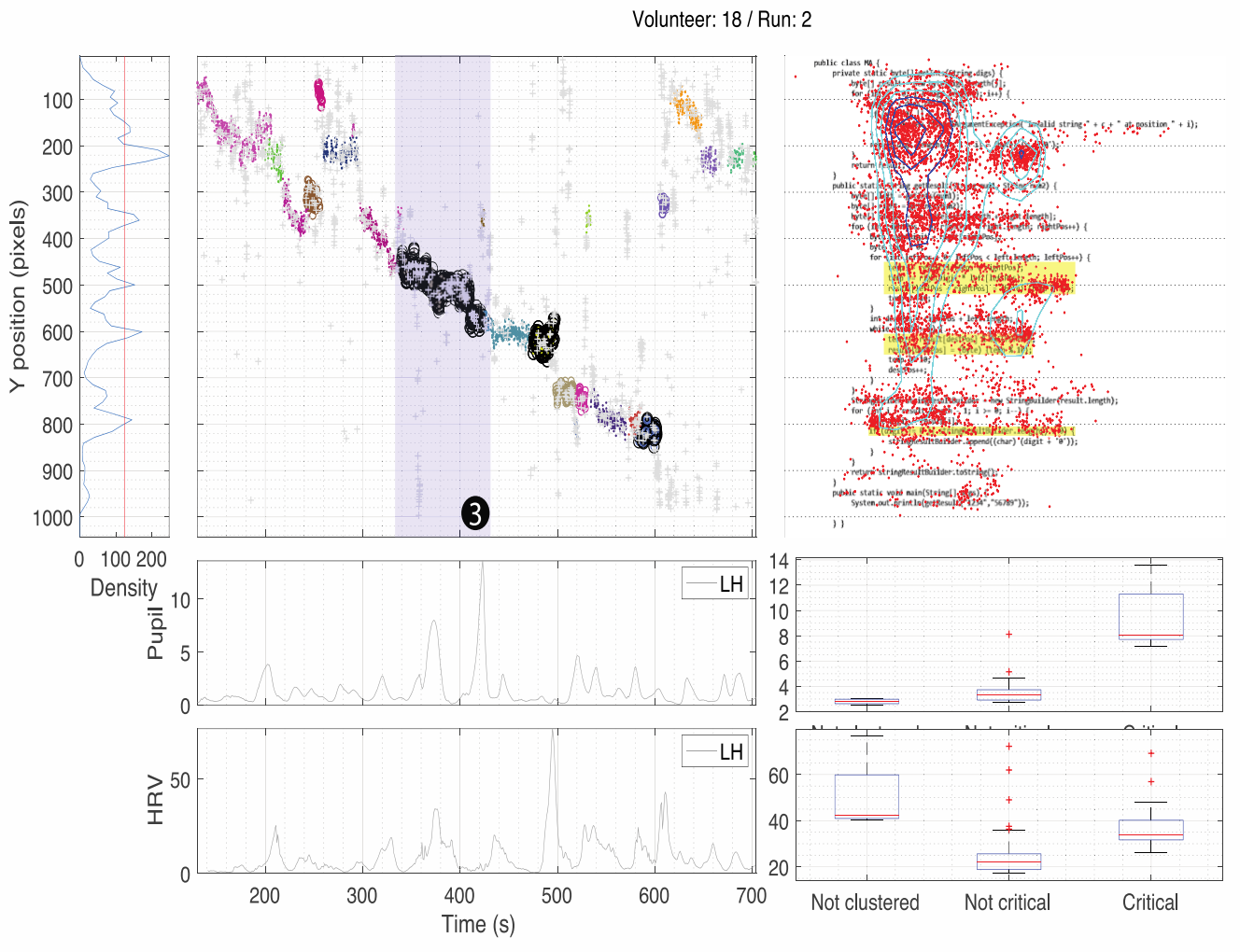

Figure 2 - Eye tracker and HRV results achieved in experience for Subject 18, code C2

Upper row/left – density of gaze line number; Upper row/middle – clusters of gaze line over time and gaze velocity; Upper row/right – red dots: geometrical distribution of gaze points superimposed on code during experience; yellow rectangle potentially critical regions of inspected code manually annotated by experts.
Middle row/left – Extracted pupilography feature LH; Middle row/right – box plot of pupilography outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions.
Lower row/left – Extracted HRV feature LH; Lower row/right – box plot of HRV outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions

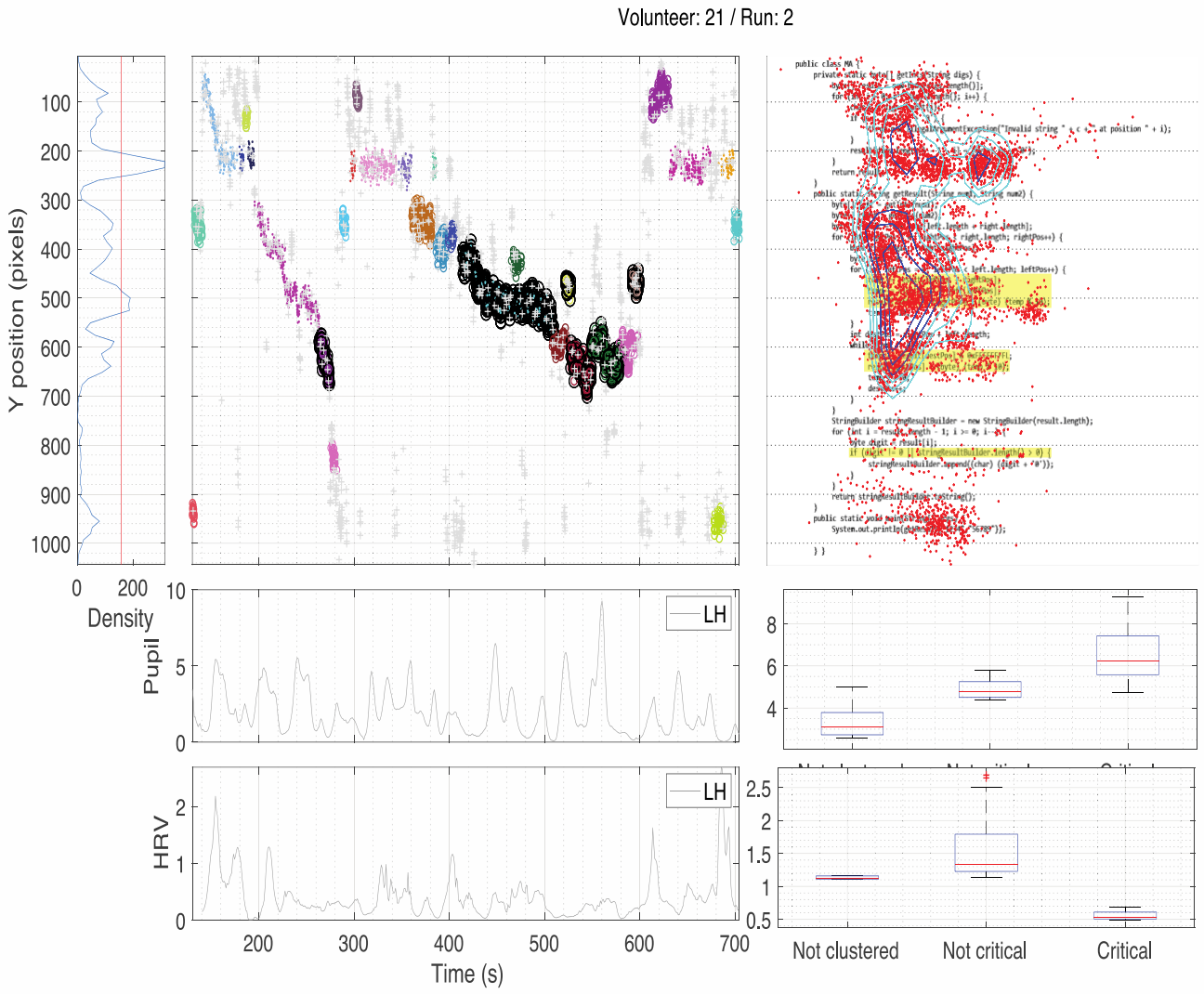

Figure 3 - Eye tracker and HRV results achieved in experience for Subject 21, code C2

Upper row/left – density of gaze line number; Upper row/middle – clusters of gaze line over time and gaze velocity; Upper row/right – red dots: geometrical distribution of gaze points superimposed on code during experience; yellow rectangle potentially critical regions of inspected code manually annotated by experts.
Middle row/left – Extracted pupilography feature LH; Middle row/right – box plot of pupilography outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions.
Lower row/left – Extracted HRV feature LH; Lower row/right – box plot of HRV outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions

the code. The four examples illustrate typical time evolution of observed gaze and physiological signals for participants during experiments using code C1 (Figure 1) and code C2 (Figures 2-4). No example for code C3 is provided due to space constraints and due to the fact that it exhibits very similar patterns compared to those observed for code C2.

Table 1 – Effort and load measured using the NASA-TLX;

Prog	OE		ME		TP		DI	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std
C1	0,96	0,11	0,29	0,11	0,3	0,15	0,27	0,12
C2	0,43	0,27	0,78	0,17	0,65	0,25	0,68	0,2
C3	0,48	0,33	0,81	0,17	0,66	0,25	0,61	0,26

The distribution of gaze points in code-space throughout each experience can be observed in the upper row/right column where gaze points sampled at 10Hz have been superimposed on the actual code used in each experience. In this graph, critical code regions identified by independent experts, i.e., not involved in the experience, have been marked in yellow. The geodesic lines correspond to the cluster of highest density of gaze points in pixel coordinates (x,y). For reference, the density of these points projected on the y – coordinate is shown in the upper row/left column.

The graph in the upper row image/middle column exhibits the evolution over time of the y-coordinates of the gaze points in order to illustrate the distinct code reading path patterns applied by each subject. The gaze points have been clustered using the feature space defined by the y-coordinate, time and code reading velocity. The rational of using code reading velocity was that it is expected that more complex sections of code should exhibit lower reading velocities compared to less complex sections of the code. The different clusters identified in this graph correspond to contiguous gaze points in time and space of low reading velocity. Gaze points marked in gray correspond to high velocity code reading sections. In order to obtain the exhibited clustering results a density-based spatial clustering algorithm has been applied.

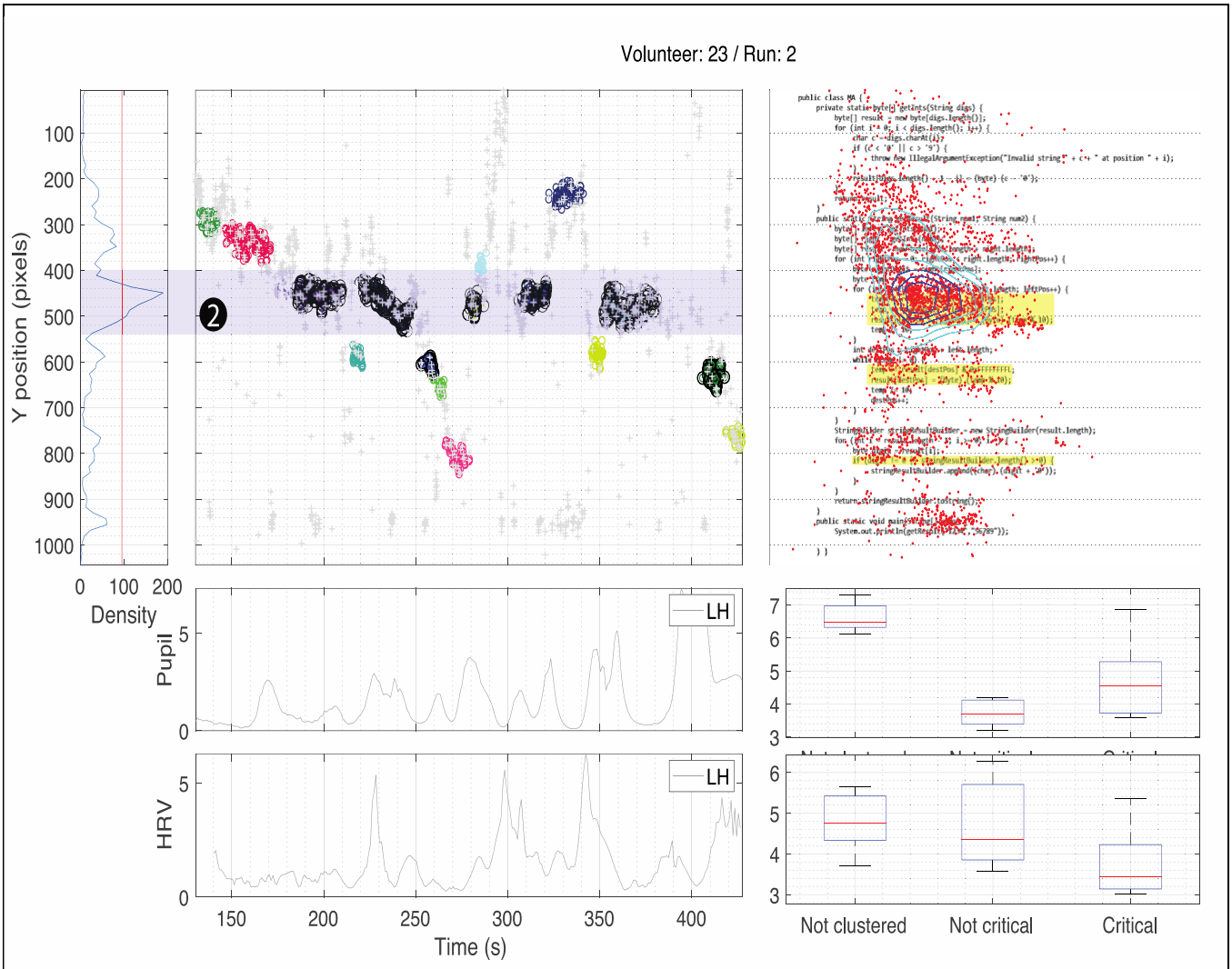

Figure 4 - Eye tracker and HRV results achieved in experience for Subject 23, code C2

Upper row/left – density of gaze line number; Upper row/middle – clusters of gaze line over time and gaze velocity; Upper row/right – red dots: geometrical distribution of gaze points superimposed on test code during experience; yellow rectangle potentially critical regions of inspected code manually annotated by experts.
Middle row/left – Extracted pupilography feature LH; Middle row/right – box plot of pupilography outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions.
Lower row/left – Extracted HRV feature LH; Lower row/right – box plot of HRV outlier feature LH in three distinct regions: not clustered gaze points, clustered gaze points in non-critical regions and clustered gaze points in critical regions

The evolution throughout the experience of the cognitive stress related features is depicted in the middle /lower row graphs positioned in the left column. Pupil LH and HRV LH have been explained previously. The LH ration in both physiological signal capture the imbalance between the sympathetic and parasympathetic of the ANS which is known to be highly sensitive to cognitive stress [41].

Finally, graphs located in the middle/lower rows/right exhibit the box-plots of the peaks of the LH features captured from pupil variability and HRV, respectively, using a 95% threshold for outlier definition.

Figures 1-4 clearly show that individual programmers' features/styles and perceived code complexity have a determinant impact on the code reading style. As was already mentioned, code C1 is much simpler compared to code C2 (this was confirmed using NASA's TLX test) and, as was already mentioned, participant 18 had no major difficulty in understanding code C2 (assessed through a survey with questions about the program). As can be observed in graphs in Figure 1 and Figure 2 upper row/middle column, in both cases the reading pattern is reasonably linear, i.e., the reader tends exhibit low number of re-visits of the same code sections, whereas in the other two cases (Figures 3 and 4) shown the reading pattern has a much higher level of re-visits of code sections, which is a typical eye tracking feature to identify relevant regions suggesting high programmers' cognitive load. This observation suggests that this feature might be used to assess code comprehension complexity. Furthermore, one might speculate whether it might be sensitive enough to assess proficiency level of programmers with relevant applications in, e.g., recruitment.

One striking observation is that both LH features exhibit complementary patterns, i.e. in critical code regions, these features tend to exhibit spikes in amplitude, yet with some delay to each other. For example, in Figure 1 a spike in the

pupil LH feature is observed between instant t=150sec and t=165sec (see marker □ in Figure 1)) in which corresponds to gaze y-coordinate that are aligned with the critical code section identified by the expert's manual annotation. The HRV LH spike is delayed by approx. 15 sec. This delay might be due to the same physiological reasons as the delay observed in the BOLD signal in fMRI studies [16, 17, 18], i.e., the hemodynamic response tends to exhibit some lag compared to other nervous modulated physiological activations. This has to be further investigated to confirm this possible explanation for the delay of HRV when compared to pupillography. It should be mentioned that these two features have both discrimination power between "not clustered", "non critical" and "critical" sections of the code as can be observed in their respective box-plots (all differences are statistically significant). Nevertheless, they are complementary and should be considered both in a multi-parametric solution for code annotation, as they might be used to reduce false positives, LH pupil spikes without corresponding HRV LH spikes and vice-versa tend to be false positives.

One of the main observations in these graphs is that the LH pupil spikes tend to occur when the reader is reading critical code regions. For example, in Figure 4 one of the critical code regions corresponds to y-coordinate range between 400 and 540 pixels (see marker □ in Figure 4)). Whenever the reader's gaze is in the aforementioned range LH spike is observed. A similar pattern is observed for most participants, although different participants may have specific critical regions related to their specific difficulties in code understanding (e.g. syntax difficulties). The achieved box-plots clearly show that these features enable discriminant power between different critical and non-critical regions in the code and therefore exhibit space (code line or token) discrimination resolution as required in code annotation applications. Furthermore, as can be observed, e.g. in Figure 2 (see marker □ in Figure 2) time resolution is also achieved since, as expected, the spike does not have the same duration as the cluster. Indeed, cognitive load might be experienced when trying to understand some code details, but once these have been grasped, unraveling the meaning of the remaining tokens of the section might be relatively easy.

This analysis of 4 examples presented in the paper is representative of the general scenario observed in all the participants. To conclude and summarize the analysis, we can refer two aspects:

1) The experiment confirm an expected result that simply consist of attesting that eye tracking, although not totally accurate, has enough resolution to pin point specific code lines that correspond to moments when HRV and pupillography shows high levels programmers' cognitive load. This is exactly the first requirement to associate programmers' cognitive load to specific code lines through biofeedback annotation.

2) The box-plots in all the four figures show that the different strategies used to define critical regions for analysis (using eye tracking metrics and defining regions using expert analysis) lead to regions that can be strongly discreminated by HRV and pupillography, suggesting that eye tracking can provide much more than just the coordinates of the code lines correspoonding to cognitive load spikes.

VI. BIOFEEDBACK CODE ANNOTATION

The analysis of experimental data in section V shows that although different programmers follow specific code reading and code comprehension strategies, the conjugation of HRV, pupillography, and eye tracking allows the identification of specific code lines that correspond cognitive load peaks from the individual programmers, suggesting that code annotation at line and token level is viable. Obviously, more detailed analysis and further studies must be performed to gain additional confidence in this preliminary conclusion. In any case, we anticipate in this section a possible organization of biofeedback annotation.

Collecting biofeedback signals during programming tasks provide us with a classifier of the cognitive load experienced by programmers. Based on this classifier, we introduce the basic concepts of *biofeedback highlighting* of code and *biofeedback-driven software testing*.

The output of the cognitive load classifier is collected throughout programming tasks and every portion of code is annotated with an integer value holding the cumulative prediction of the risk of that code containing a software defect. Each source file has a twin file containing the annotations.

Every lexical token (identifiers, keywords, operators, *etc.*) in a source file is annotated with the cumulative risk predictor. Lexical tokens are the symbols recognized by compilers during lexical analysis [38] and are also the smallest units of code that can be added, modified or deleted by programmers. Thus, annotating every token provides us with the level of detail to use the biofeedback classifier of cognitive load. For example, on input

```
while (y != x) {
      y = f(x);
}
```

the sequence of tokens (each composed by a token name, an optional token value and the biofeedback annotation) is

<while,0> <(,0> <id,"y",0> <!=,0> <num,"0",0>
<),0> <{,0> <id,"y",0> <=,0>
<id, "f",0> <(,0> <id,"x",0> <),0> <;,0> <},0>

All annotated values are set to zero, to exemplify a situation in which the biofeedback classifier always returned zero (i.e., no risk of software defects. Supposing that a programmer modifies the condition to (y != x && y >= 1) and the classifier yields a value of 4, then the token stream is annotated accordingly, resulting in

<while,0> <(,0> <id,"y",0> <!=,0> <num,"0",0>
<&&,4> <id,"y",4> <>=,4> <num,"1",4>
<),0> <{,0> <id,"y",0> <=,0>
<id, "f",0> <(,0> <id,"x",0> <),0> <;,0> <},0>

Using the lexical token as the unit of annotation provides us with the adequate granularity, because the same annotation may be used to label any portion of code from a single line (all tokens have the same value) or a square region involving multiple lines. Furthermore, as a programmer writes code, the sequence of modifications may involve single tokens in different lines of code.

A. Biofeedback highlighting

Programmers carry out lightweight inspections and walkthroughs to improve code quality. The concept of biofeedback highlighting consists in coloring the tokens presented by an IDE according to the biofeedback code annotation. This complements the classical syntax

highlighting by providing visual cues for those reading the code to know the portions of code with a greater risk of containing defects.

To achieve this, the sequence of additions, modifications and deletions performed on the code are recorded with a event-logging plugin and, in parallel, the programmer's biofeedback is collected and classified. The output of the classifier is associated to the lexical tokens most recently added or modified by the programmer (using the sliding window method earlier described in the paper).

B. Biofeedback-driven software testing

The biofeedback annotation is stored in a twin file associated with each source code file. Given that software testing is one of the most costly activities in software engineering projects, the concept of biofeedback-driven software testing consists in focusing the testing process on the parts of the source code that have a greater risk of defects.

To this end, studies have shown that a software defect most often consists of a single mistake or few mistakes in a single function [4, 39]. Given a software testing budget consisting of a number of test cases, one is able to improve the coverage of the testing process by focusing on the functions that are globally classified and annotated with a higher risk of containing defects. To do so, the classical coverage criteria (source line coverage, control-flow testing, data-flow testing, etc.) are complemented with the need to provide coverage to the statements and functions that contain lexical tokens annotated with a high biofeedback classification value. The end result is a higher coverage of the portions of code that were subjectively perceived as the most difficult, which is a complement to the classical coverage criteria that objectively classify the code with respect to some metric of complexity.

VII. THREATS TO VALIDITY

As a controlled experiment, the study reported in this paper has several limitations concerning the realism of the setup and the natural limits of a complex experiment with people. Although our goal is to assess the possibility of doing accurate code annotations expressing programmers' cognitive load while dealing with specific code lines or lexical tokens (i.e., we are not reporting yet a ready to use technology), it is worth discussing the most relevant threats to validity of our study, which are summarized in the following paragraphs.

The code samples are not representative of large software products. In fact, as explained before, the three programs were carefully designed to create three different scenarios concerning code complexity. Even so, the size of the programs is rather modest when compared to actual code. The limitation here is the reasonable time that can be assigned to the volunteers for code comprehension, which results in relatively small programs. Although this is a problem for the realism of the experiment, since the key point is to evaluate the possibility of identifying specific code lines that are perceived by the programmers as more difficult, we believe that the small scale of the programs used in the experiment does not affect the validity of the results.

The number of volunteers (30) is an obvious limitation. Although the number 30 is normally associated to the boundary that provides robust statistical analysis, we are aware that the personalized nature of the biofeedback process recommends a larger sample. In fact, different volunteers having different expertise will tend to have difficulties in different locations of the code, which means that the global analysis would benefit from a larger number of volunteers.

The analysis reported is really a first step analysis focused on checking that critical code regions (previously defined by experts) can be identified using pupillography and HRV features in most of the volunteers. More complex analysis including the clustering of the different types of volunteers (Intermediate, Advanced, and Expert) is being carried out to confirm the conclusions reported in this paper. Definitive conclusions will require the use of the proposed biofeedback code annotation technique in real software development setups.

VIII. CONCLUSION

Information extracted from ANS activity has already been shown to have the potential to be used to distinguish among different levels of perceived code complexity. In practice, this result does not allow for code line/token annotation, as required to provide feedback to programmers on possible buggy lines or to guide the testing effort. In this paper, the goal was to evaluate the feasibility of using physiology related information sources modulated by ANS activity to perform code line/token annotation using a multi-parametric approach. The goal was to show the feasibility of using this approach in order to achieve adequate space-time resolution in the identification of complex regions of code that could benefit from a second look in order to potentially reduce bugs.

A study was conducted using instrumented programmers during code comprehension tasks of different complexity levels. We showed that the imbalance feature between the sympathetic and parasympathetic ANS activity extracted from the HRV and the pupillography shows significant potential to 1) detect cognitive load induced by reading critical code sections, which can be identified at code line level of detailed using eye tracking and 2) that HRV and pupillography combined with gaze information captured with the eye tracker has the potential to provide the targeted space-time resolution for code annotation. This multi-parametric approach is inherently person-specific as code comprehension difficulties are highly dependent from person to person. The results suggest that a person-specific approach should be followed.

The eye–tracker provides an immensely rich information source that has still to be fully explored in programming tasks. E.g. the feasibly study reported herein suggests that recurrence in code viewing is an indicator of perceived code complexity / programmer proficiency.

Future work will be focused on proving the potential unveiled by the present feasibility study by developing a fully automated programmer-specific multi-parametric approach that is able to autonomously annotate code during coding and testing activities. As was mentioned, there are multiple open issues that still have to be fully researched in order to allow this step. In this paper we identified several open research questions that will contribute to this goal.

Acknowledgment

The authors would like to thank to the volunteers that participated in the experiments. This work was partially funded by the BASE project, POCI - 01-0145 - FEDER-031581, and also partially supported by the Centro de Informtica e Sistemas da Universidade de Coimbra (CISUC).

REFERENCES

[1] C. Wohlin, “Is there a Future for Empirical Software Engineering?”, Proc.of the 10th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement, Sept., 2016.

[2] C. Amrit, M. Daneva, D. Damian, “Human factors in software development: On its underlying theories and the value of learning from related disciplines; A Guest Editorial Introduction to the Special Issue”, Information and Software Technology journal, Sept. 2014.

[3] James Reason, “Human Error”, Cambridge University Press, 1990.

[4] A. Fuqun Huang, B. Bin Liu, and C. Bing Huang, “A Taxonomy System to Identify Human Error Causes for Software Defects”, 18th ISSAT Int. Conf. on Reliability and Quality in Design, 2012.

[5] Fuqun Huang, “Human Error Analysis in Software Engineering”, in the book “Theory and Application on Cognitive Factors and Risk Management”, F. Felice and A. Petrillo Eds., IntechOpen, 2017.

[6] R. Couceiro, G. Duarte, J. Durães, J. Castelhano, C. Duarte, C. Teixeira, Miguel C. Branco, P. Carvalho, H. Madeira, “Biofeedback augmented software engineering: monitoring of programmers' mental effort”, Int. Conference on Software Engineering, New Ideas and Emerging Results, ICSE 2019.

[7] R. Couceiro, G. Duarte, J. Durães, J. Castelhano, C. Duarte, C. Teixeira, Miguel C. Branco, P. Carvalho, H. Madeira, “Pupillography as indicator of programmers’ mental effort and cognitive overload”, IEEE Int. Conf. on Dependable Systems and Networks – DSN 2019, USA, June 24 – 27, 2019.

[8] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates SW Faults", Proc. of the 26th International Fault Tolerant Computing Symposium, FTCS-26, Sendai, Japan, 1996.

[9] J. Durães and H. Madeira “Emulation of SW Faults: A Field Data Study and a Practical Approach”, IEEE Transactions on SW Engineering, vol. 32, no. 11, pp. 849-867, November 2006.

[10] R. Natella, D. Cotroneo, J. Duraes, H. Madeira, "On Fault Representativeness of SW Fault Injection", IEEE Transactions on SW Engineering, vol.39, no.1, pp. 80-96, January 2013.

[11] V. Anu, et. Al, “Using A Cognitive Psychology Perspective on Errors to Improve Requirements Quality: An Empirical Investigation”, IEEE 27th International Symposium on Software Reliability Engineering, 2016.

[12] T. Nakagawa, et al., “Quantifying Programmers’ Mental Workload During Program Comprehension Based on Cerebral Blood Flow Measurement: A Controlled Experiment”, Proc. of ICSE 2014.

[13] B. Floyd, T. Santander, and W. Weimer, “Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise”, ICSE 2017, pp 175–186, Piscataway, NJ, USA, 2017.

[14] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, “Understanding understanding source code with functional magnetic resonance imaging”, Proc. of the 36th International Conference on Software Engineering - ICSE 2014.

[15] N. Peitek, J. Siegmund, et al., “A Look into Programmers' Heads”, IEEE Transactions on Software Engineering, August, 2018.

[16] J. Castelhano, I. C. Duarte, C. Ferreira, J. Durães, H. Madeira, and M. Castelo-Branco, “The Role of the Insula in Intuitive Expert Bug Detection in Computer Code: An fMRI Study”, Brain Imaging and Behavior, 2018.

[17] S. C. Müller and T. Fritz, “Using (Bio)Metrics to Predict Code Quality Online”, Department of Informatics, University of Zurich, Switzerland, Proc. of 38th IEEE ICSE, 2016.

[18] E. Hess and J. Polt, "Pupil size in relation to mental activity during simple problem-solving," Science, vol. 143, pp. 1190-1192, 1964.

[19] J. Beatty, "Task-evoked pupillary responses, processing load, and the structure of processing resources," Psychological bulletin, vol. 91, 1982.

[20] J. Beatty and B. Lucero-Wagoner, "The pupillary system," Handbook of psychophysiology, vol. 2, 2000.

[21] S. Sirois and J. Brisson, "Pupillometry," Wiley Interdisciplinary Reviews: Cognitive Science, vol. 5, pp. 679-692, 2014.

[22] M. L. H. Võ, et al., "The coupling of emotion and cognition in the eye: Introducing the pupil old/new effect," Psychophysiology, vol. 45, pp. 130-140, 2008.

[23] H. Lüdtke, B. Wilhelm, M. Adler, F. Schaeffel, and H. Wilhelm, "Mathematical procedures in data recording and processing of pupillary fatigue waves", Vision research, vol. 38, pp. 2889-2896, 1998.

[24] M. Nakayama and Y. Shimizu, "Frequency analysis of task evoked pupillary response and eye-movement," in Proceedings of the 2004 symposium on Eye tracking research & applications, pp. 71-76, 2004.

[25] A. Murata and H. Iwase, "Evaluation of mental workload by fluctuation analysis of pupil area," in Engineering in Medicine and Biology Society, 1998. Proceedings of the 20th Annual International Conference of the IEEE, pp. 3094-3097, 1998.

[26] V. Peysakhovich, M. Causse, S. Scannella, and F. Dehais, "Frequency analysis of a task-evoked pupillary response: Luminance-independent measure of mental effort," International Journal of Psychophysiology, vol. 97, pp. 30-37, 2015.

[27] U. Obaidellah, M. Haek, P. Cheng, “A Survey on the Usage of Eye-Tracking in Computer Programming”, ACM Comp. Surveys, 2018

[28] Z. Sharafi, T. Shaffer, B. Sharif, Y-H Guéhéneuc, “Eye-Tracking Metrics in Software Engineering”, Asia-Pacific Software Engineering Conference, Dec. 2015.

[29] B. Sharif, H. Kagdi, “On the use of eye tracking in software traceability”, Proc. International Conference on Software Engineering, Jan. 2011.

[30] T. Busjahn, C. Schulte, B. Sharif, M. A. Jetbrains, “Eye Tracking in Computing Education”, International Computing Education Research Conference, August 2014.

[31] I. McChesney, R. Bond, “Eye tracking analysis of computer program comprehension in programmers with dyslexia”, Empirical Software Engineering, Springer, Sept. 2018.

[32] D. Kondrashov and M. Ghil, "Spatio-temporal filling of missing points in geophysical data sets," Nonlinear Processes in Geophysics, vol. 13, pp. 151-159, 2006.

[33] R. Sassi, V. D. Corino, and L. T. Mainardi, "Analysis of surface atrial signals: time series with missing data?," Annals of biomedical engineering, vol. 37, pp. 2082-2092, 2009.

[34] F. Onorati, M. Mauri, V. Russo, and L. Mainardi, "Reconstruction of pupil dilation signal during eye blinking events," in Proceeding of the 7th International Workshop on Biosignal Interpretation, pp. 117-120, 2012.

[35] M. Nakayama and Y. Shimizu, "Frequency analysis of task evoked pupillary response and eye-movement," Proc. of 2004 symposium on Eye tracking research & applications, pp. 71-76, 2004.

[36] A. Eleuteri, A. C. Fisher, D. Groves, and C. J. Dewhurst, "An efficient time-varying filter for detrending and bandwidth limiting the heart rate variability tachogram without resampling: MATLAB open-source code and internet web-based implementation," Computational and mathematical methods in medicine, vol. 2012.

[37] Burg, J.P. "Maximum Entropy Spectral Analysis", Proc. of the 37th Meeting of the Society of Exploration Geophysicists, 1967.

[38] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, 2nd edition, Addison-Wesley, 2006.

[39] R. Barbosa, F. Cerveira, L. Goncalo and H. Madeira, Emulating representative software vulnerabilities using field data, in Computing, doi.org/10.1007/s00607-018-0657-y, Springer, August, 2018.